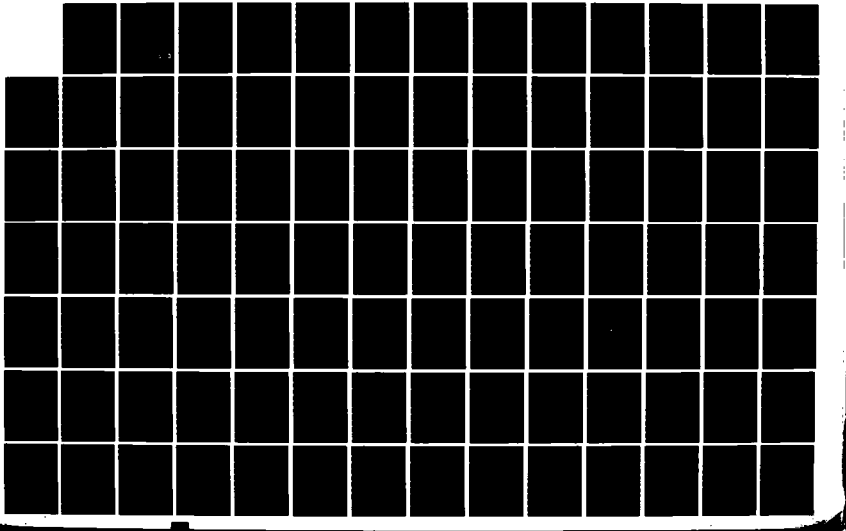
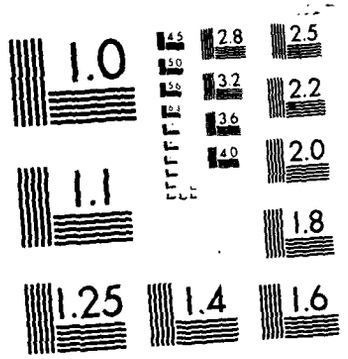
AD-A156 905 GENERATION OF FLIGHT PATHS USING HIERARCHICAL PLANNING (U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH K B KLINE 1985 AFIT/CI/NR-85-37T 1/3

UNCLASSIFIED F/G 12/2 NL





GENERATION OF FLIGHT PATHS

USING HIERARCHICAL PLANNING

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

KEVIN BRIAN KLINE
B.S., Pennsylvania State University, 1979
A.S., Thomas Nelson Community College, 1977

1985
Wright State University

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER AFIT/CI/NR 85-37T	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Generation Of Flight Paths Using Hierarchical Planning		5. TYPE OF REPORT & PERIOD COVERED THESIS/DISSERTATION 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Kevin Brian Kline		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS AFIT STUDENT AT: Wright State University		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS AFIT/NR WPAFB OH 45433		12. REPORT DATE 1985 13. NUMBER OF PAGES 120
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASS 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

APPROVED FOR PUBLIC RELEASE: IAW AFR 190-17

LYNN E. WOLAVER
Dean for Research and
Professional Development
(4 May 8) AFIT, Wright-Patterson AFB OH

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

ATTACHED

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASS

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

ABSTRACT

Kline, Kevin Brian. M.S., Department of Computer Science, Wright State University, 1985. Generation of Flight Paths Using Hierarchical Planning.

This thesis examines the use of an artificial intelligence technique, hierarchical planning, to solve the problem of generating an aircraft route and finding a path through various hostile environments. A route or path, is evaluated by the number and type of threats the aircraft encounters on the route and the route length. An algorithm using hierarchical planning is presented and tested against several hostile environments. Specifically, the algorithm will divide the problem space or grid, into smaller spaces or boxes. These boxes are then assigned values based upon the input hostile environment. Block paths are then constructed and evaluated based on the values in the boxes. An exhaustive search is performed on the two best block paths to find a flight path for the aircraft. Test results are compared to previous results obtained using heuristic search and indicate an improvement in solution quality. Although specific plans are incorporated into the algorithm to obtain test results, many other plans within the realm of hierarchical planning certainly exist and could be used to solve this problem.

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

March, 1985

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Kevin Brian Kline ENTITLED Generation of Flight Paths Using Hierarchical Planning BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE .

Thomas A. Sudkamp
Thesis Director

Chairman of Department

Committee on
Final Examination

Thomas A Sudkamp

Dean of the School of Graduate Studies

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

LIST OF FIGURES (CONTINUED)

LIST OF TABLES

To Peg, Scott, Chris, and Chalene

The most important people in my life for their love and support through the years.

I. INTRODUCTION

The objective of this thesis is to study the effectiveness of hierarchical planning to construct a route or path for an aircraft through a hostile environment. An evaluation of a path is based on the number and type of threats an aircraft will encounter on the path. When more than one path has the same cost, the shortest path is considered to be the better path. The use of hierarchical planning for this problem was suggested by Dr. Thomas Sudkamp, Department of Computer Science, Wright State University. A thesis recently submitted under Dr. Sudkamp's supervision [5], approached the same problem using heuristic search. Results from the hierarchical planning method will be compared to the same test cases used in the heuristic search approach.

This paper will define the problem environment and other methods that are used to solve it. The hierarchical planning model will be described along with the planning strategy and how it was developed. The algorithm along with its implementation is presented and test cases with their results will be discussed. Concluding remarks concerning the hierarchical planning method for solving this problem will then be presented.

II. PROBLEM ENVIRONMENT

The problem is to generate an aircraft route through a hostile environment. The problem space is represented by a grid with all points of the grid designated by x and y coordinates. The origin of the grid is in the lower left corner and is designated by the x, y coordinates (0, 0). The aircraft route is defined as starting at the leftmost side of the grid, moving left to right, and ending at the rightmost boundary. A path from a start position to the goal consists of several flight legs. A flight leg is the part of a path connecting two points. The entire grid represents a possible hostile environment. A threat is represented by a circle either partially or entirely within the grid. Each threat location is given by x, y coordinates which indicate its center. There are different threat types, each designated by two components: radius and cost of the threat. The cost of a threat is the cost assigned to a path which intersects the threat. The total cost for a path is the sum of all the costs for the threats the path encounters. Actual computation of the threat cost will be covered in the Hierarchical Planner (HPLAN) implementation. As an example, define a grid with maximum x, y coordinates as (100, 50). The aircraft route or path, is shown as a line starting at (0, 25) and ending at (100, 25). The threats and their costs are given below in Table 2.1. The grid, threats, and path are illustrated in Figure 2.1. The cost of the path shown is 85. This is calculated by adding the values of the threats the path encounters along the way. In this case, the path passes through

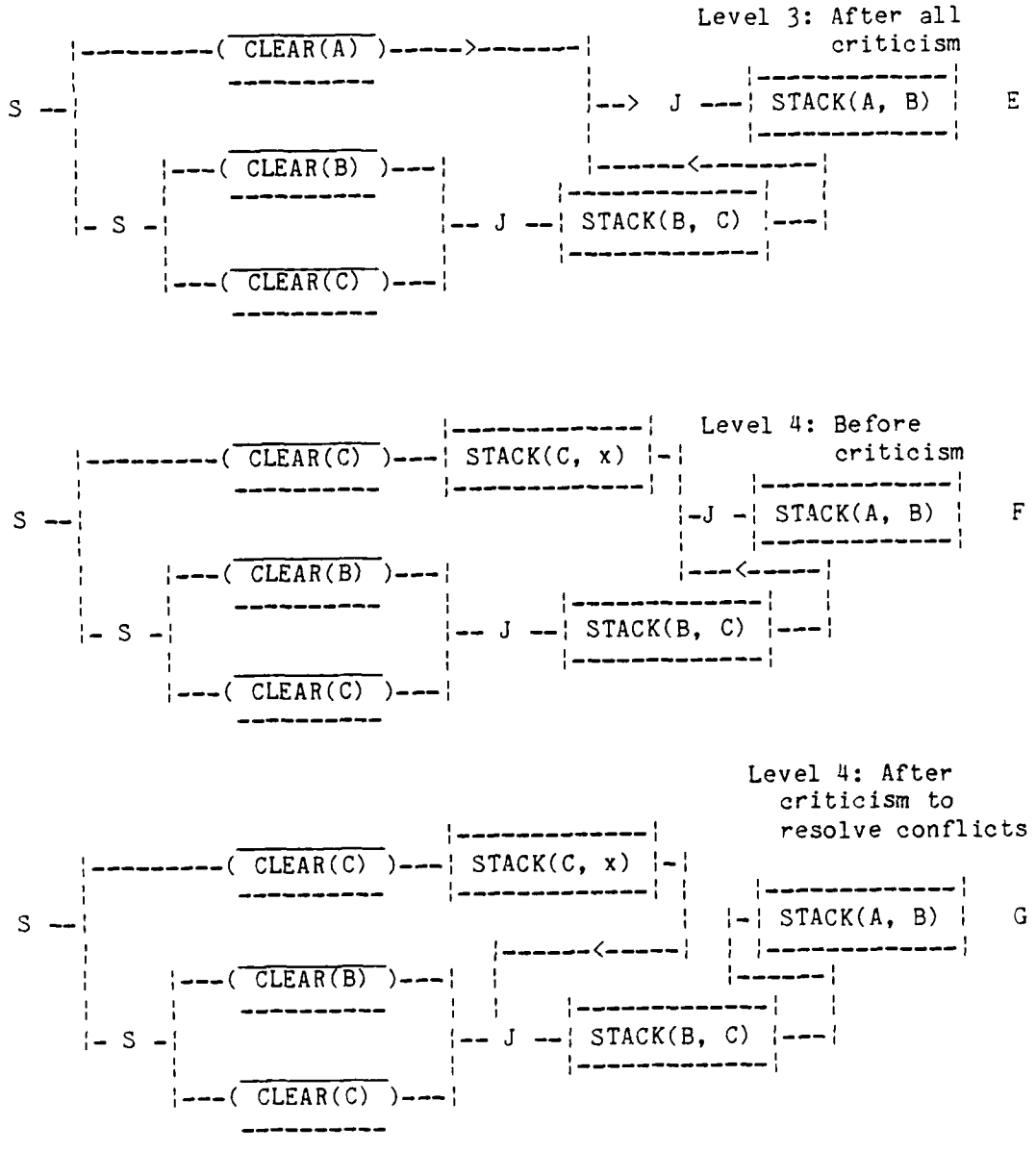


Figure 4.2 -- continued

NOAH Plans for Example Problem

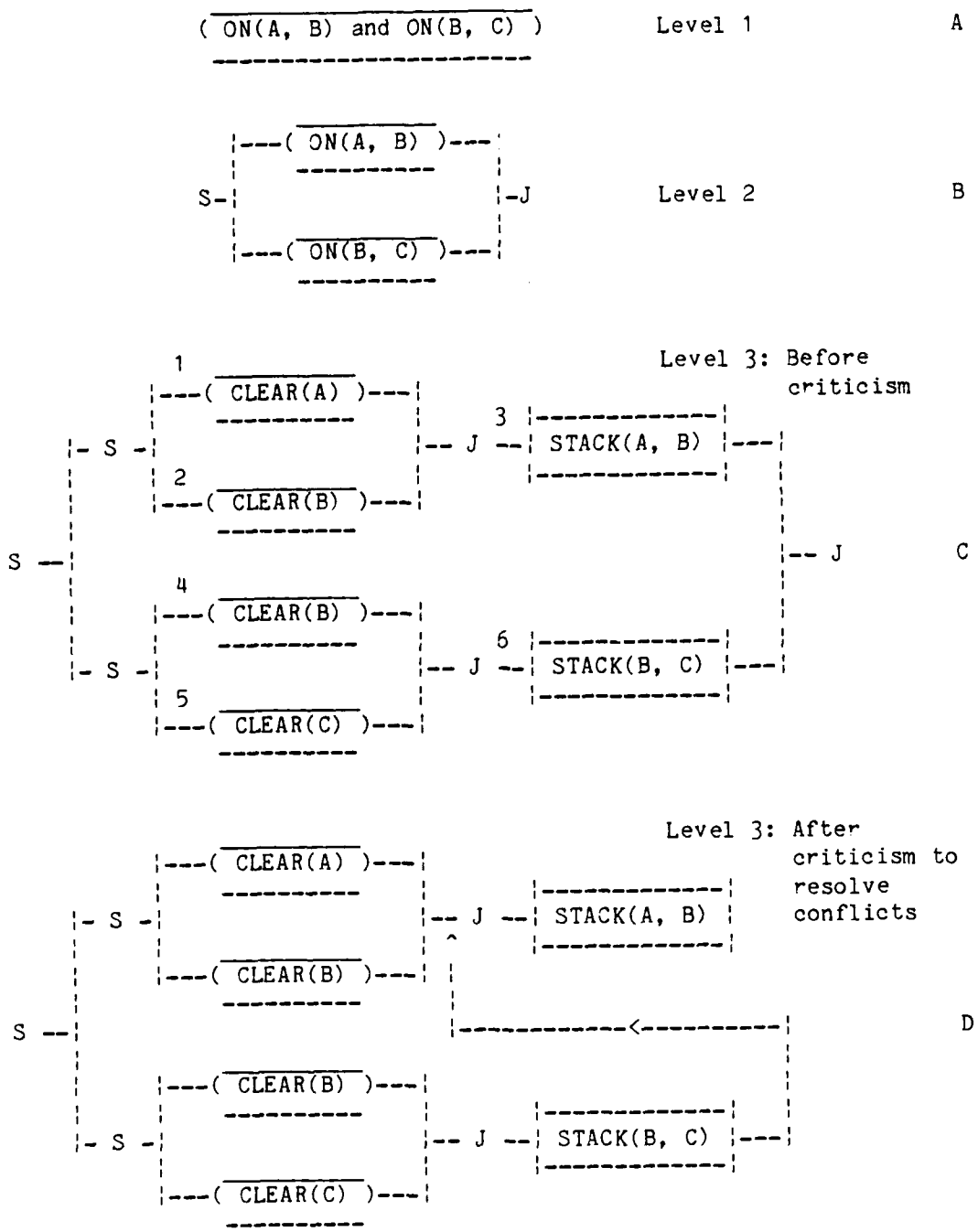
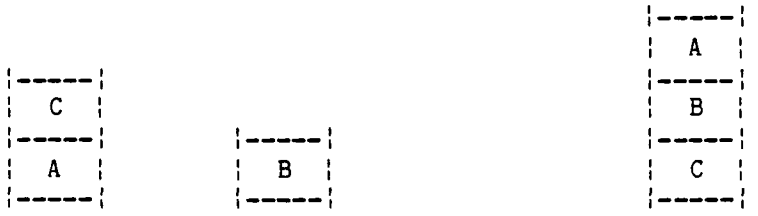


Figure 4.2

NOAH Plans for Sample Problem

graph D. The top CLEAR(B) is redundant because the precondition must exist only for the eariler action, STACK(B, C). This results in graph E. Graph F results from observing that in order to CLEAR(A), C must be removed from A and C must be clear to do that. The Resolve Conflicts critic is used again to ensure that everything depending on C being clear was done prior to STACK(B, C). The result is graph G. The Eliminate Redundant Preconditions critic is used to delete one CLEAR(C) and results in graph H. Next the system observes that CLEAR(C) and CLEAR(B) are true in the initial state. Therefore, the final plan produced by NOAH is in graph I.



Initial State:

ON(C, A) and

CLEAR(B).

Goal:

ON(A, B) and

ON(B, C).

Figure 4.1

Sample Blocks Problem

the problem presented in Figure 4.1.

The first thing NOAH does is to divide the problem (graph A) into two subproblems (graph B). The preconditions of STACK are now considered and shown in graph C. The nodes in graph C are numbered to aid in explaining actions of the critics. Now the critic, Resolve Conflicts, is invoked and a table is constructed showing literals that must be true for one operation but are also denied by some operation. The table entries are presented below:

```
CLEAR(B):    asserted:  node 2 "CLEAR B"
             denied:    node 3 "STACK A on B"
             asserted:  node 4 "CLEAR B"
CLEAR(C):    asserted:  node 5 "CLEAR C"
             denied:    node 6 "STACK B on C"
```

Sometimes something must be true before an operation is performed, but then will be denied by that same operation. For example, CLEAR(C) denies the STACK(B, C) operation, but the CLEAR(C) protects STACK(B, C). Therefore, those preconditions that are denied by the operation they are protecting are deleted from the table and results in the following table:

```
CLEAR(B):    denied:    node 3 "STACK A on B"
             asserted:  node 4 "CLEAR B"
```

This entry contains the fact that since STACK(A, B) will undo the precondition for STACK(B, C), STACK(B, C) should be done first. Graph D shows the plan after this criticism. Now another critic, Eliminate Redundant Preconditions, is invoked. The goal CLEAR(B) appears twice in

detailed actions. An example described in [9] and [11] is given to show how NOAH would solve a problem. The problem domain is the blocks world. The blocks world consists of a number of square blocks, all the same size, that can be stacked on each other, and a flat surface on which blocks can be placed. There is a robot arm that can move the blocks. The actions that it can perform and predicates required to specify certain conditions, that are needed for this example are:

ON(A, B):	Block A is on block B.
CLEAR(A):	There is nothing on top of block A.
STACK(A, B):	Will put block A on block B, provided that both objects are clear.

The blocks problem for this example is shown in Figure 4.1. In the initial state, block C is on top of block A and block B is on the table by itself. The final goal is to have block A on block B and also have block B on block C.

NOAH uses a structure called a procedural net, which is a graph structure whose nodes represent actions at varying levels of detail, organized into a hierarchy of partially ordered time sequences. Nodes labeled S indicate a split in the plan and nodes labeled J indicate a join. Square boxes represent operators that will be incorporated into the plan. Boxes with rounded ends denote goals that remain to be satisfied. NOAH also uses a set of "critics" to examine plans and interactions between the subplans. Two examples include critics that are used to resolve conflicts in plans and eliminate redundant specifications of subgoals. Use of these critics will become clear in the discussion of the example. Figure 4.2 illustrates how NOAH solves

details of the original problem are introduced and this process continues until the original problem is solved. Sacerdoti [10] terms this process as "length-first" search. This is because the method uses an abstraction space and follows through to the original goal state before exploring the next level of abstraction or computing the final solution. This enables the program to find any steps that will possibly lead to dead ends or undesirable solutions.

The technique of hierarchical planning can be illustrated by an example of finding a suitable travel route from Dayton, Ohio to San Antonio, Texas. Instead of being forced to consider the details such as route numbers, consider the subproblem of going from the state of Ohio to Texas. In order to ignore further details at this time, plot the route from Ohio to Texas by going state to state. Now the subproblem to be solved is to find a route from Dayton, Ohio to San Antonio, Texas through Indiana, Illinois, Missouri, and Oklahoma. This process may continue to solve additional subproblems or go directly to the final solution. The main point of this example is the total search space, all routes in the United States, has been reduced to just the routes that are contained in the solution for the subproblem. Additionally, the details of looking at individual routes was not introduced immediately, but ignored until they are needed.

The problem-solving system NOAH [11] also performs planning in a hierarchical fashion. It does this by first constructing an abstract skeleton of a plan and then successively considers more details. NOAH is presented a problem as a statement that is to be made true by applying a sequence of actions to a given initial state. Then it introduces another level of detail of the solution and solves for more

IV. THE HIERARCHICAL PLANNING MODEL

Hierarchical planning is a techinique that divides a problem into smaller subproblems, with the important capability to defer consideration of the details of a problem at the higher level plan. The crucial point is to discriminate between the important information and details of the problem space. Details of the problem are essentially ignored at the higher level plan or plans and are only introduced when the subproblems are solved. This approach delays consideration of some details until the problem search space has been significantly reduced by solving the subproblems at a higher level. The details of the problem are accounted for after a solution is found in an abstract problem space, thereby increasing program power and efficiency.

THE HIERARCHICAL PLANNING TECHNIQUE

For complicated problems, one technique used is to work on small pieces of the problem and combine their solutions into a complete solution for the original problem. The planning approach involves decomposing or breaking down the original problem into appropriate subparts. Decomposition involves dividing the original problem into several subproblems. In the hierarchical planning technique, the original problem space is redefined into one or more abstraction spaces. Sacerdoti [10] describes an abstraction space as a simplifying representation of the problem space in which unimportant details are ignored. These abstraction spaces can then be searched to solve simple, more manageable subproblems. As these subproblems are solved, more

contain weights that may be adjusted. The problem is in obtaining the right combination of weights. A problem encountered with this method is computer run time relating to leg length. As leg length is decreased, the total number of legs generated is increased. Because actual costs are computed and heuristic evaluation for an estimate of future costs is required for each leg, computer time is increased significantly. For example, based on the results in [5], when leg length was reduced by one half, CPU time increased by three to one hundred times.

Comparing the two models above, SNOOPER looks at the total threat environment to determine the best path but has a computational explosion because it examines every possible move. FPG is only concerned with evaluating a single leg at a time and therefore does not take advantage of future threat analysis. One possible improvement is to use a technique to cut down on the search space while taking into account the entire threat environment. This leads to the next method discussed, heuristic search.

HEURISTIC SEARCH MODEL

The heuristic search model [5] defines the problem space in much the same way as the two previous methods. There is a grid defined with an x and y axis and threats located throughout the grid. The model attempts to find a solution to the problem by using a modified A* search algorithm. An A* algorithm is a method that uses heuristics, knowledge applied to an algorithm to minimize search space, to direct the path search in an 'intelligent' manner. The goal is to limit the number of paths the algorithm must search through and at the same time find the best path. Paths consist of legs which are generated based on user input. These inputs define the path deflection angle, which is equivalent to the threat detection wedge in the FPG model, the length of a leg, and the number of legs in the path detection angle. The important feature of this method are the heuristics. Heuristics are used to estimate the future costs to the goal. The methods of cost computation are detailed in [5]. This method uses heuristics to cut down on the total search space and determine a path based on the heuristics. The actual makeup of the heuristics is paramount with this method. Poor heuristics may produce bad results. The heuristics used

and number of legs from a point are also aircraft-dependent. Together these parameters define the look-ahead capability called the threat detection wedge. Figure 3.2 illustrates an aircraft's threat detection

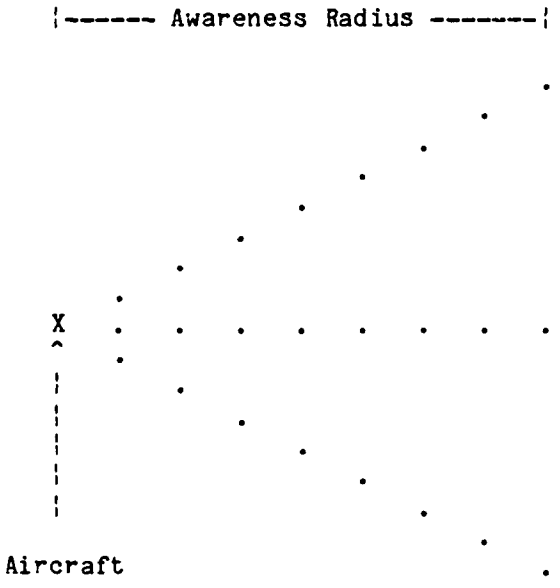


Figure 3.2

Threat Detection Wedge

wedge. The algorithm looks at the possible legs from the current point and determines which leg is best. This determination is made by finding the leg that will encounter the least amount of threat. Threats outside of the awareness wedge are not considered. In other words, future threats on the grid are not considered when determining a current leg. This process continues until the goal state is reached. This model is useful when there is no prior knowledge of the entire threat environment, but undesirable compared to other methods when all the threats are known because it cannot anticipate future threats in the grid.

illustrated in Figure 3.1. Legs between adjacent points incur a cost or

1 2 3

4 X 5

6 7 8

Figure 3.1

8 Possible Legs to X

penalty. The cost is calculated based on the distance between points, the altitude and speed of the aircraft, and threat interaction. The results are saved and used to determine the best path from the goal to the desired starting point. The best path or route is defined as the one that encounters the least amount of threat. Length is also used to determine the best path, but is secondary to the threat cost of the path. This method will find the best path because it examines all possible moves from each point defined in the grid, but this exhaustive search of all the possible points is restrictive because of the computer resources it requires.

FLIGHT PATH GENERATION (FPG) MODEL

The FPG model [4] also uses a grid defined by an x, y axis. A start state and goal state are defined and the algorithm begins at the start state. The distance from the start state to the goal state is a path consisting of several legs. The length of each leg is equal to the distance a particular aircraft can look ahead to see threats. This distance is called the awareness radius of the aircraft. The direction

III. SOLUTION METHODS

This section describes three solution methods for finding a suitable aircraft route through hostile environments. The first two methods described are the Strategy Needed Over Optimum Penetration Routes (SNOOPER) model [6], and the Flight Path Generation (FPG) model [4]. Although only briefly discussed, these two models have influenced the third method which in turn provided an invaluable reference to this thesis. The third method referred to above, Generation of Flight Paths Using Heurisic Search [5], was a thesis recently submitted at Wright State University and will be discussed in depth because it uses another artificial intelligence technique, heuristic search. Another solution method, hierarchical planning, is used in this thesis and will be described in a later chapter.

STRATEGY NEEDED OVER OPTIMUM PENETRATION ROUTES (SNOOPER) MODEL

The SNOOPER model [6] defines the problem space as rectangular in shape and locations in this grid are referenced by x, y coordinates. Within this grid, a point is specified as the goal state. This method searches backward, from the goal state to an undefined starting state. There also exists threat sites within the grid which affect the flight path of the aircraft. The model begins at a fixed goal point on a grid and examines all possible moves to that point and continues this process for each point generated. Penetration route movement from points in the grid proceed to any of eight surrounding points. Movement is either parallel to the x or y axis or at a 45 degree angle. This is

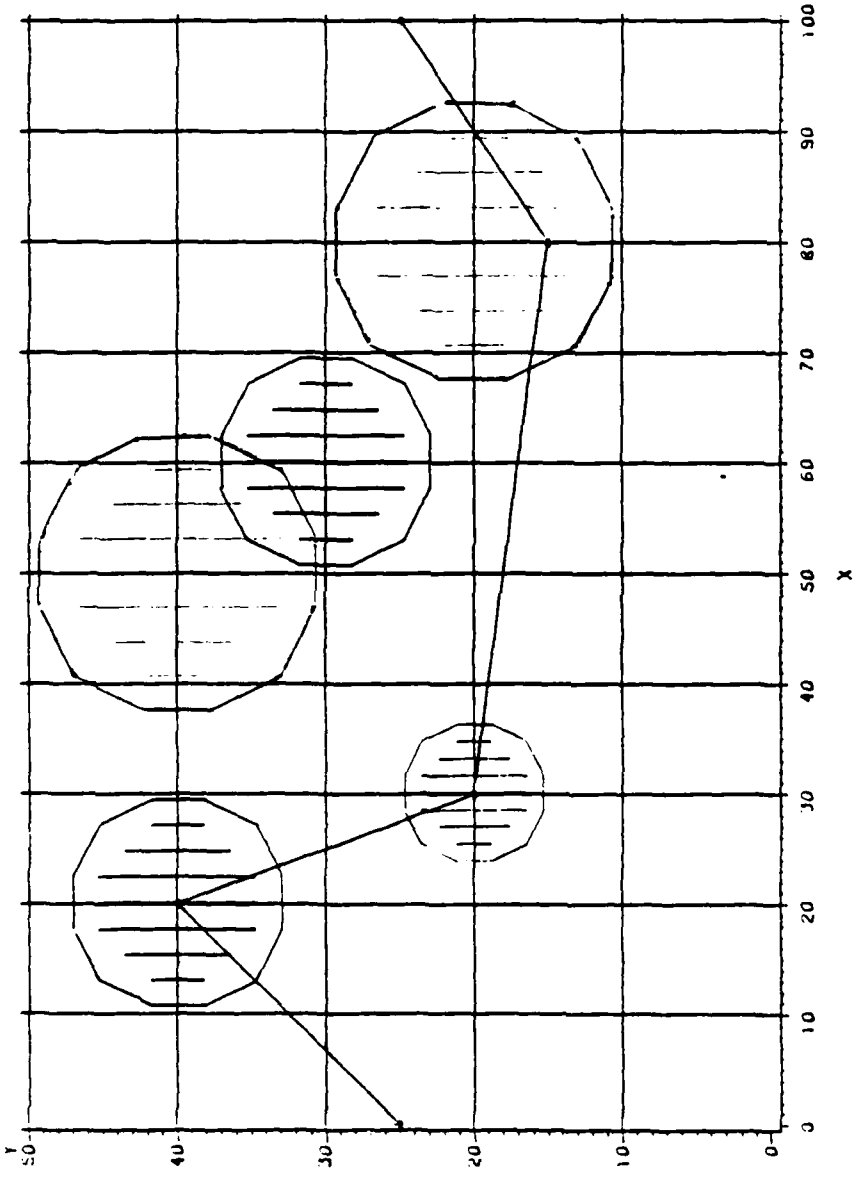


Figure 2.1

Sample Grid, Threats, and Path

threat numbers 1, 3, and 4. The total of the threat costs is 85, which is the cost of the given path.

Threat #	X, Y	Radius	Cost
1	30, 20	5	50
2	50, 40	10	10
3	80, 20	10	10
4	20, 40	7.5	25
5	60, 30	7.5	25

Table 2.1

Sample Threat Values

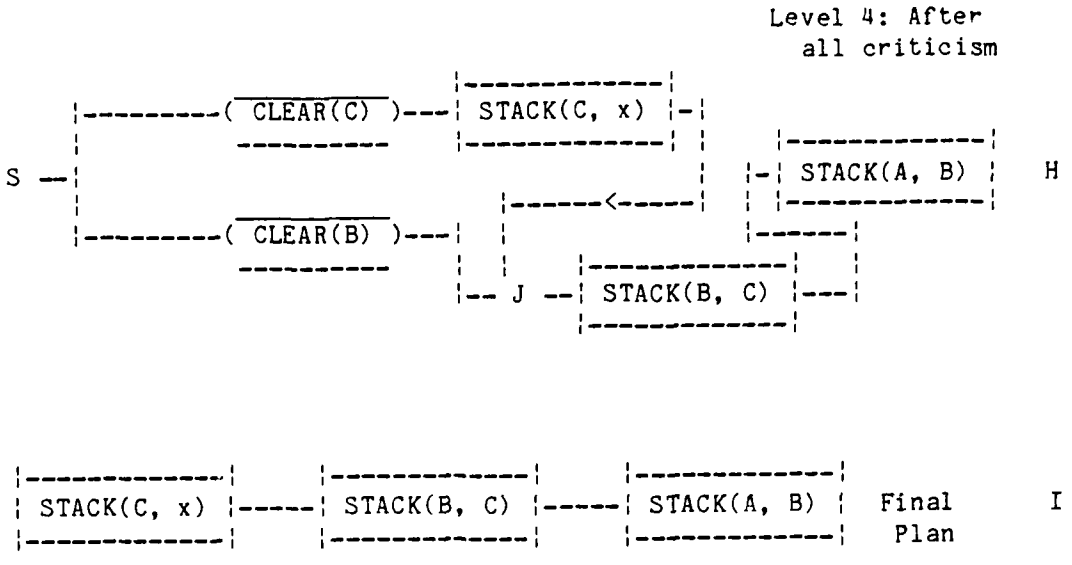


Figure 4.2 -- concluded

NOAH Plans for Sample Problem

MOTIVATION FOR HIERARCHICAL PLANNING

Some computer programs are written to generate all possible solutions and test these solutions to discover the best solution. Many problems are nontrivial and the number of possible solutions and search space are extremely large. Because of computer restraints, such as CPU time and memory, no exhaustive search of the entire problem space is possible. This indicates a need to direct the search through the problem space in order to reduce computer time and memory requirements. Although an improvement over an exhaustive search of the entire problem space, even heuristic search can result in a combinatorial explosion. As noted in [3], even with using heuristic functions, it becomes apparent that planning must be incorporated if substantial reductions in search effort are to be achieved. Nontrivial problems contain many details that need to be taken into consideration in solving the problem and finding the best solution. Attention to these details is exactly the reason for the drawbacks of many problem solvers. The solution is to try to eliminate part of the search space and reduce the original problem to a smaller one and put off consideration of as many details as possible, for as long as possible. This approach would search through an abstraction space, a simplifying representation of the problem space, in which certain details are ignored. When a solution is found to a subproblem in the abstraction space, then details are considered and a final solution to the original problem is found.

The problem solving technique examined in this thesis will be hierarchical planning. This method is used for several reasons. One reason was to observe the similarities and differences between hierarchical planning and heuristic search methods in the same problem

environment. Results from test cases using hierarchical planning will be presented and compared to the results of the heuristic search method obtained in [5]. Another reason this approach to problem solving is used is because it has displayed significant increases in problem-solving power in other systems. The ABSTRIPS problem-solver [10] progressively narrows the search space by solving the problem using simpler, less constrained operators. By solving these simpler problems, Sacerdoti [11] notes that ABSTRIPS is relatively insensitive to combinatorial explosion. In the NOAH system [11], consideration of details in a stepwise fashion increases efficiency because many details of the problem are solved only at lower-level plans. The method of generalizing plans in [2], replacing problem-specific constants with problem-independent parameters, increases its problem-solving capabilities.

V. DETERMINATION OF PLANS

Given the the hierarchical planning method and the problem environment, the next decisions to make are what planning strategy should be used to divide the problem space into subproblems and which details could be ignored at the higher level plans. The first decision made was to divide the problem space into boxes and use an evaluation method to assign values to each box. After this, use a technique to reduce the overall grid size, repeating the process until the search space is small enough to use exhaustive search to find a set of potential solution paths. The details of the problem, such as actual path cost and final path legs, would be considered only after the original search space was small enough to perform the exhaustive search. A decision was made not to use heuristics in this process for two reasons. First, thesis research for this problem using heuristic search was ongoing. Secondly, it was decided that the design should attempt to reduce the search space through hierarchical planning, so that exhaustive search techniques could be applied.

In order to compare final results with those used in the heuristic search method and to keep the problem space consistent throughout the testing phase, the grid is defined to be 100 units wide and 50 units high with starting point at (0, 25) and ending point at (100, 25). Given this grid size, a determination would now be made on how to subdivide the problem space. The grid would be divided into 10 X 10 unit boxes. Now the grid is 10 boxes wide and 5 boxes high. These boxes

would then be used to form block paths from the start position to the goal. Figure 5.1 shows how the grid is divided into boxes. Boxes marked with an X illustrate an example block path.

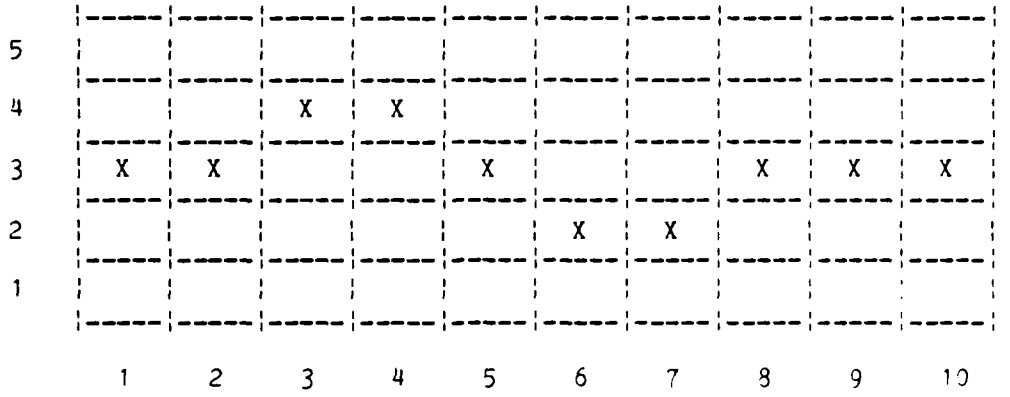


Figure 5.1

Sample Grid and Block Path

Various plans are included in the final version of HPLAN. Although these will be discussed in detail in the next section, they will be discussed briefly now. Different methods for calculating box values in the block paths are used in HPLAN. First, if a threat center is in a box, that box gets the threat value. In addition, any box that has its center included in that threat's circle also receives the threat value. Another method uses the same technique for computation of box values as above, but does so with smaller 5 X 5 boxes. Once these smaller boxes have values, four of them are added together to obtain a value for the big 10 X 10 box that contains them. A third method using small boxes considers only the eight surrounding boxes as possible candidates for the threat value. The major improvement in the final design of HPLAN is the way in which the best block paths are determined. Since the search space is smaller, an exhaustive search is performed to

find all possible block paths. The best block paths are those that contain the least amount of threats, based on the values in the boxes. In the final design of HPLAN, there are two methods for calculating total block path cost. One method simply adds the values for all the boxes in a block path for the block path cost. The other method adds only the values of boxes in its block path that have not aready been added from the same threat. In other words, if a threat covers two adjacent boxes, and both boxes are contained in the block path, the value of only one box will be added for that threat. This eliminates counting the same threat twice in the same block path. Differences between the two methods will be noted later.

After determination of the best block paths, the block path now becomes the new, smaller search space for the problem. Now an exhaustive search is feasible because the search space is much smaller. An exhaustive search is performed within a block path to locate the best flight path. It is at this lower level that details of the original problem can be introduced. Calculating actual flight path costs through random threats, determining the best flight path from beginning to end, and calculating path lengths are details that were ignored at the higher level plans but are now considered. When this is accomplished, the finalized flight path with cost and length computed, is found and the original problem is solved.

VI. HPLAN IMPLEMENTATION

The hierarchical planning algorithm (HPLAN) is implemented in the programming language PASCAL. Program coding and testing were performed on both DEC VAX 11/750 and VAX 11/780 mini-computers. Graphics were designed and produced on the IBM 3083 system using SAS/GRAPHICS and a CALCOMP 1012 plotter. The following sections will describe the basic components of the HPLAN implementation. First, the HPLAN algorithm will be described in general terms to explain the program flow. Following this, block path construction and methods for computing box values will be discussed. The next sections describe block paths and flight paths. The final sections describe and illustrate HPLAN input and output.

HPLAN ALGORITHM

The HPLAN program generates aircraft routes and finds a path through random hostile environments using hierarchical planning. The algorithm begins reading an input data file containing program options and a set of random threats. Program options will be explained throughout the following sections. The set of threats in the input file define the hostile environment the aircraft must fly through. Each threat is defined by the x, y coordinates of its center, a radius to define how far the threat extends, and a value to represent its cost. After the data is read, the program processes the threats. This includes storing threat values and computing individual box values in the grid. Once all the boxes defined in the grid have a value assigned, block paths containing these boxes are generated. All possible block

paths are examined to determine which ones are the best candidates for an exhaustive search to find the best flight path. Two block paths are chosen and an exhaustive search is then performed on both. Each exhaustive search produces the best possible flight path within the block path. The two flight paths are compared and the one that has the least cost will be the final path selected. Should both flight paths incur the same cost, the shorter flight path is considered best. Finally, the best flight path, its block path, program options and threat values are output. The program flow just discussed is shown in Figure 6.1. The entire program listing for HPLAN is presented in Appendix A.

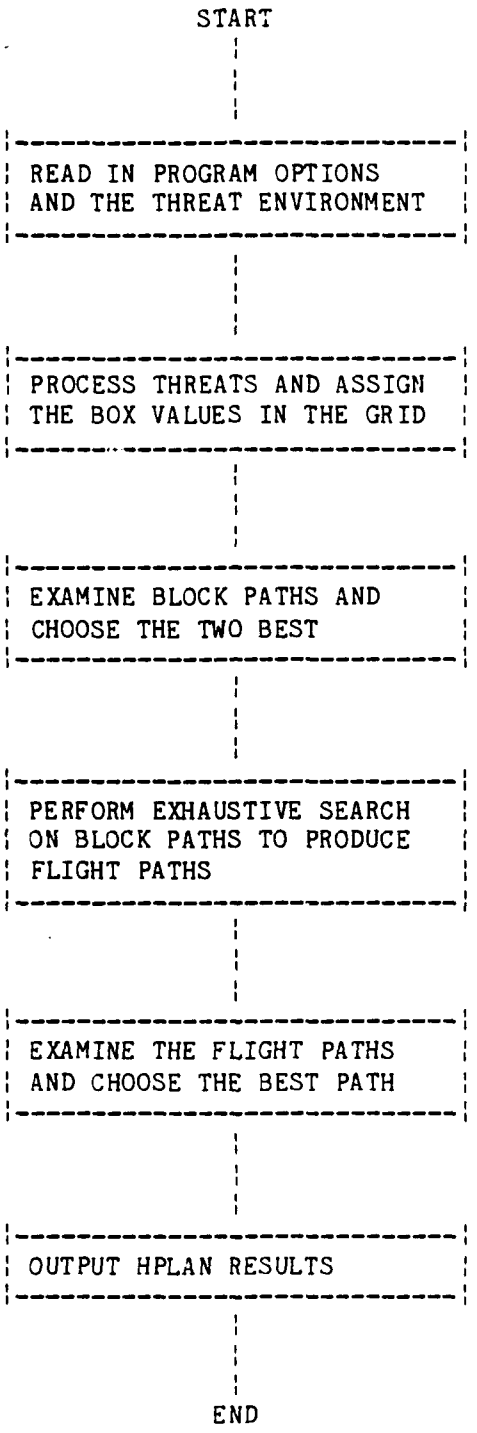


Figure 6.1

Program Flow for HPLAN

BLOCK PATH CONSTRUCTION

HPLAN is developed to allow for several processing options. A set of options defines a plan to produce a flight path through a threat environment. Options are considered throughout the algorithm and dictate several processing decisions in HPLAN. HPLAN options exist for describing how the grid is divided and how block paths are built. The grid size is 100 units wide and 50 units high. To describe how the grid is divided, x and y values are input to define how many boxes the grid will contain. For example, if x is 10 and y is 5, the grid will be 10 boxes along the x-axis and 5 boxes along the y-axis. Therefore, the grid will contain 10 X 10 unit boxes. Similarly, if x is 20 and y is 10, the grid will contain 5 X 5 unit boxes. The size of the boxes in the grid will either be 5 X 5 or 10 X 10 depending on the x and y values.

To define how block paths are built, the type of path is input. If the type of path is BIG, block paths will be constructed with 10 X 10 boxes. If the type of path is SMALL, the block paths will consist of 5 X 5 boxes. The x and y values and the type of path together defire three plans for block path construction depending on the combinations selected. If the type of path selected is BIG, then x and y may be either 10,5 or 20,10 respectively. The BIG/10,5 combination indicates the final block paths are 10 X 10 boxes with box values computed based on the 10 X 10 boxes. The BIG/20,10 combination again defines the final block path as 10 X 10 boxes, but box values are first computed using the 5 X 5 boxes and are then added to obtain values for the 10 X 10 boxes. The third plan these combinations produce is the SMALL/20,10 option. This defines block paths with 5 X 5 boxes with box values computed using 5 X 5 boxes.

BOX VALUES

Values assigned to boxes in the grid are calculated differently depending on the HPLAN options selected. After the threat environment is input to HPLAN, each individual threat is processed to determine which box in the grid contains the threat center. This box is then assigned a value equal to that threat's cost value. Since a threat may extend over more than one box, a check is made to determine what other boxes are affected by the same threat. The other boxes checked for containment within the threat are determined by the box coverage option. If the option is TOTAL, then all boxes within the threat radius are considered. If the box coverage option is CENTER, then only those eight boxes that are adjacent to the box containing the threat center are considered. The center option is only available when computing values for small 5 X 5 boxes. In either case, those boxes considered are assigned the threat cost if the threat contains the center point of a box, in which case we say that the box is covered by the threat. Boxes that are covered by more than one threat add the threat costs together to compute the box value. This is illustrated in Figure 6.2. Box A is assigned the threat cost but box B is not because the center of box B is not contained in the threat. Box C is assigned the sum of the two threat costs because the center of box C is contained in both threats. Boxes D and E will be assigned the threat cost because both their centers are contained in the threat. The exception to this is when the cost option DECONFLICT is selected. This method will store the threat costs for a box separately for block path score computation. The reason for storing the values separately will be explained in the next section.

This process continues for each threat. If the grid is divided

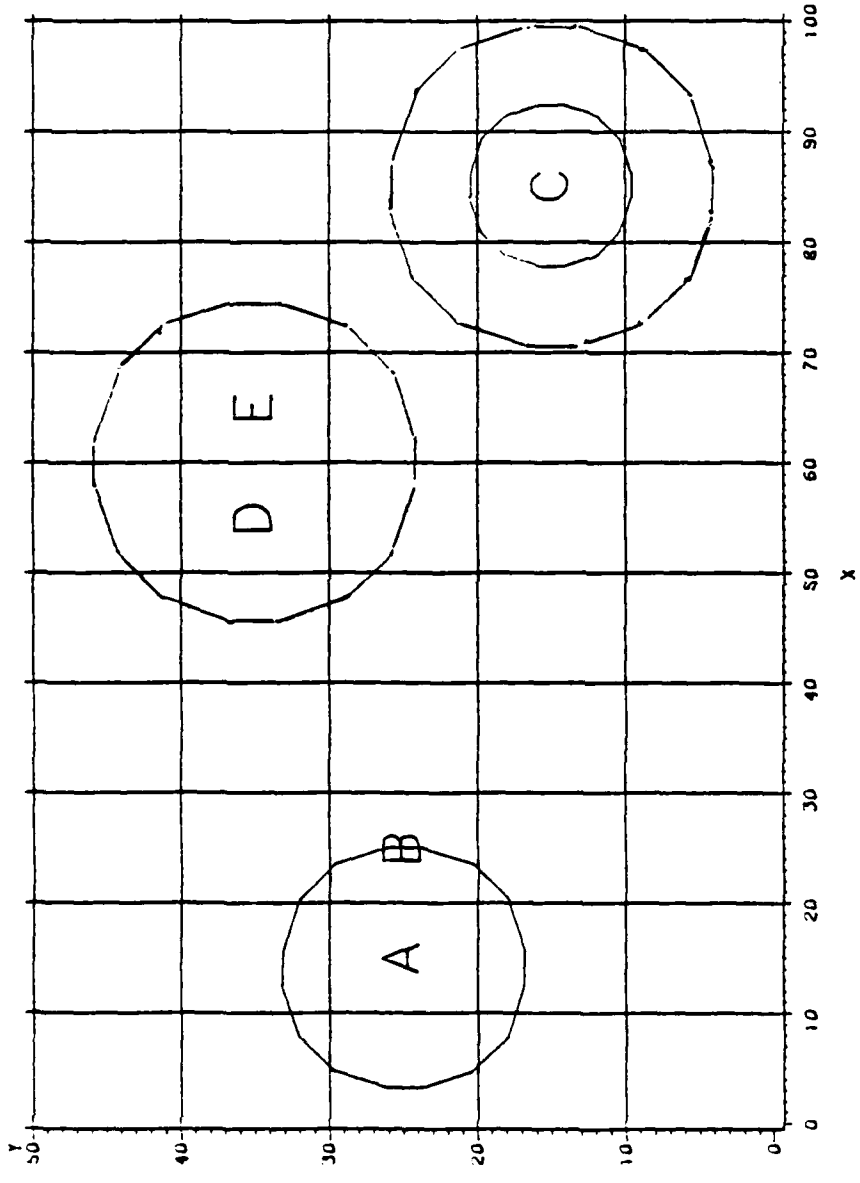


Figure 6.2

Box Values

into 200 small boxes, but the type of block path is BIG, indicating 10 X 10 boxes, values in the small 5 X 5 boxes are added to obtain values for the big 10 X 10 boxes. Each 10 X 10 box contains four 5 X 5 boxes. The values from these four boxes are added together to obtain the value for the 10 X 10 box. Boxes in the grid now have values to indicate the threat cost assigned to each box.

BLOCK PATHS

The next process after all boxes have values assigned, is to generate block paths and evaluate them to determine the best block paths. The quality of the final solution depends on this process. A block path is a path from the starting point on the grid to the goal point via boxes in the grid. If the path option in HPLAN is BIG, then the block paths consist of 10 X 10 boxes, otherwise 5 X 5 boxes are used. First, all possible block paths in the grid are generated. Possible block paths are those that contain the starting point (0, 25) and continue through adjacent boxes until the block path reaches the box containing the goal point (100, 25). This process begins with the box containing the starting point (0, 25) and uses recursion to build all possible block paths through the entire grid. The next possible box in a block path from the current box are those that continue the path in a forward direction towards the goal box. From a current box, there are at most three boxes for the next move in the block path. This is illustrated in Figure 6.3. If box A is the current box, then the next box in the block path must be B, C, or D.

Since the block path must be "driven" towards the goal box (the box containing the goal point), several boxes in the grid are not candidates for a possible block path. A box may be a valid candidate

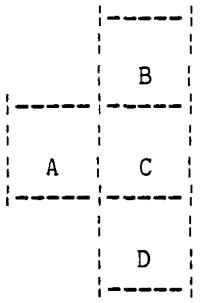


Figure 6.3

Next Box Moves

because it continues the path in a forward direction, but it may not be considered because it cannot possibly lead to the goal box. Figures 6.4 and 6.5 illustrate both BIG and SMALL block paths and indicate with an 'X' those boxes in each grid that are not possible candidates to produce a valid block path. In Figure 6.4 for example, box 9, 5 is not a candidate for a possible block path because only boxes 10, 5 and 10, 4 can be reached from it and they cannot reach the goal box by valid moves. Therefore, when generating the next possible move from box 8, 4, box 9, 5 is not a possible move because the goal box cannot be reached.

As each possible block path is generated, a score is assigned to the block path. Since all boxes now have values assigned to them, each block path can be assigned a score equal to the values in the boxes in the block path. Lower scores indicate better block paths. The method for determining a block path score is given by the cost option, ALL or DECONFLICT. The ALL option calculates the score by adding all the values in the boxes that are contained in the block path. For example, if there are 10 boxes, each with a value of 10, the final score for the block path is 100. The DECONFLICT option also adds values from the boxes, but may not add all the values. In the previous section

appoximately 54 hours of CPU time. In contrast, CPU time using big block paths never exceeded 8 minutes and was as little as 9 seconds. Tests 3 through 8 contain more run options because these were considered final test cases for the heuristic search results. All possible combinations of HPLAN options are tested against the last five test cases. The number of points per division used for each option are 5, 9, and 17. These were selected because they offer a varying range between the minimum and maximum points, 2 and 20. They were also selected because they successively add points exactly between each other thereby producing results at least as good as the set preceeding them.

TEST 1

The first observation from Table 7.1 is that the x, y values 20, 10 produced better results than 10, 5. Adding values from small boxes to obtain big box values works better in this case. The block paths from the two methods were different. As can be seen in Figure 7.1, the block path forced the flight path for the 10, 5 option to the top portion of the grid. Once in that area, the only way the path can reach the goal is to pass through a threat with a cost of either 25 or 40. Because threats are polygons rather than circles in the graphs, a path may seem to intersect a threat edge when in fact the path actually misses the threat. This can be resolved by checking the path cost that is listed with each graph. The block path with x and y values at 20, 10 forced the flight path to the lower part of the grid. Figure 7.2 shows that the path can reach the goal by passing through just two threats, each having a cost of only 10. Table 7.1 also shows the cost options, ALL and DECONFLICT, when x, y are 10, 5, produce the same results. Additionally, when x, y are 20, 10, the box coverage options CENTER and

VII. HPLAN RESULTS

Test results from HPLAN are presented for eight test cases. Each test case represents a different threat environment. The random threat environments used here are the same as those used in [5] to facilitate comparison between hierarchical planning and heuristic search. Results from each test are presented separately. First, a discussion of a test result is given. This includes comments concerning the different run options, discussion of block paths, flight paths, and comparison to heuristic search results. Next, the results are presented in tables. Each table will describe the threat environment which includes the random seed used to create the threats, the total threat density, the number of threats generated, and the number of threat types. For each threat type, its radius, cost, and density are given. A table entry represents a computer run and consists of the type of path, cost option, box coverage, x and y values, number of divisions, and the number of points per division. The last three columns in a table entry give the CPU time in seconds, the path cost, and the path length. The last three items are also given for the heuristic search method. Multiple runs were made in [5] on test cases and this entry represents the best path results. Finally, graphs are presented for test cases. Graphs illustrate the grid, threat environment, and the flight path.

Various program options were used on the test cases. Only two test cases (tests 2 and 3) used the type path SMALL option because of CPU time requirements. Processing time using the small block paths used

```
BOX THREAT VALUES

  5  0    50     0     0     0     0     0     0    50    50
  4 50     0    50     0     0     0    25     0    50     0
  3  0    25    50    50    50     0     0     0     0    50
  2 35    10    25     0     0     0    50   100    50     0
  1 25     0    25     0    10     0     0     0     0     0
     1     2     3     4     5     6     7     8     9    10

BLOCK PATH

          X          Y

          1          3
          2          4
          3          5
          4          5
          5          5
          6          5
          7          5
          8          4
          9          3
         10          3

SCORE =   50


FLIGHT PATH

          X          Y

       0.00      25.00
      10.00      32.50
      20.00      40.00
      30.00      40.00
      40.00      40.00
      50.00      40.00
      60.00      40.00
      70.00      40.00
      80.00      35.00
      90.00      30.00
     100.00      25.00

COST =  50.00

LENGTH = 108.54

CPU TIME(msec) =       17860
```

Figure 6.11 -- concluded

Sample Output Listing from HPLAN

HPLAN RESULTS:

OPTIONS SELECTED:

```
   TYPE OF BLOCK PATHS --  BIG
   BLOCK PATH COSTS    --  ALL
   BOX THREAT COVERAGE --  TOTAL
   XY BLOCK LIMITS     --  10  5
   NUMBER OF DIVISIONS --  10
   POINTS PER DIVISION --   9
```

INPUT THREAT ENVIRONMENT:

TOTAL NUMBER OF THREATS -- 18

THREAT INPUTS -- X Y RADIUS COST

1.92	40.26	5.0	50
90.13	31.94	5.0	50
80.31	13.91	5.0	50
83.21	42.73	5.0	50
11.14	41.33	5.0	50
65.46	17.82	5.0	50
24.10	34.22	5.0	50
29.05	25.19	5.0	50
73.87	16.81	5.0	50
91.18	45.30	5.0	50
40.15	25.81	5.0	50
94.03	21.13	5.0	50
27.62	10.46	7.5	25
4.79	8.14	7.5	25
65.97	32.45	7.5	25
15.64	24.30	7.5	25
9.88	16.02	10.0	10
43.79	0.08	10.0	10

Figure 6.11

Sample Output Listing from HPLAN

HPLAN OUTPUT

HPLAN output results reflect the input options and threat environment, matrix representing values for boxes in the grid, final block and flight paths, and selected statistics. Figure 6.11 presents a sample output listing from HPLAN. The first section lists all the HPLAN options selected and the input threat environment. The input threat environment consists of the total number of threats and the x, y coordinates, radius, and cost for each threat. A matrix is also output containing the values assigned to the boxes in the grid. If the cost option is ALL, the final values for each box are shown. If the path option is BIG with x, y coordinates 20,10, the matrix with small box values and the big box values with the small boxes added, are both shown. If the cost option is DECONFLICT, the matrix output will show the individual threat values assigned to each box. After all box values are displayed, the block path that produced the final flight path is given with its x, y coordinates and score. The flight path is output with x, y coordinates, the cost of the path, and the path length. The CPU time in milliseconds is also shown. This represents the CPU time used not including data input and output. Graphic output showing the threat environment and flight path is produced using information from the HPLAN output.

```
BIG
ALL
TOTAL
10 5
10 10
54.16    1.28    40    5.0
36.29   31.99    40    5.0
22.45   35.79    40    5.0
54.78   78.49    40    5.0
27.58   39.54    25    7.5
38.90   23.65    25    7.5
12.11   97.23    25    7.5
43.21   45.98    25    7.5
23.67   84.13    25    7.5
19.78   21.56    10   10.0
61.23   39.33    10   10.0
 3.01   30.11    10   10.0
   -1      -1    -1     -1
```

Figure 6.10

Sample DATAIN File

```
ENTER TYPE OF BLOCK PATH -- BIG or SMALL

BIG

ENTER COST OPTION -- ALL or DECONFLICT

ALL

ENTER BOX THREAT COVERAGE -- CENTER or TOTAL

TOTAL

ENTER MAXIMUM XY BLOCK COORDINATES -- XMAX YMAX

10 5

ENTER NUMBER OF DIVISIONS AND THE NUMBER OF
POINTS FOR EACH DIVISION -- NUMDIV PTS

10 10

ENTER SEED

31583

ENTER NUMBER OF THREAT CATEGORIES

3

ENTER TOTAL THREAT DENSITY

1.0

FOR THREAT CATEGORY 1 ENTER RADIUS, COST, DENSITY

5 40 0.4

FOR THREAT CATEGORY 2 ENTER RADIUS, COST, DENSITY

7.5 25 0.3

FOR THREAT CATEGORY 3 ENTER RADIUS, COST, DENSITY

10 10 0.3
```

Figure 6.9

Sample Run of THREAT_BLDR

program listing for THREAT_BLDR is presented in Appendix B. A sample of THREAT_BLDR prompts with user inputs underlined is shown in Figure 6.9. An example file produced by THREAT_BLDR and used for input to HPLAN is given in Figure 6.10.

		Cost Option	Box Coverage	X, Y	Num Div	Pts/ Div
Type Path	BIG	ALL/DECONFLICT	TOTAL	10,5	10/20	2-20
		ALL	CENTER/TOTAL	20,10	10/20	2-20
	SMALL	ALL/DECONFLICT	CENTER/TOTAL	20,10	20	2-20

Table 6.1

HPLAN Options

HPLAN INPUT

Program inputs for HPLAN consist of program options and values representing the threat environment. These values are read by HPLAN via a data file. This data file, called DATAIN, is produced by a separate program, THREAT_BLDR. THREAT_BLDR is written in PASCAL and prompts the user for HPLAN options and other information required to produce a random threat environment. The user inputs the type of block path, cost option, box coverage, maximum x and y block coordinates, number of divisions, and the number of points for each division. These inputs are stored in DATAIN and used by HPLAN for developing a specific plan. Table 6.1 lists the options available in HPLAN and their various combinations. The remaining portion of THREAT_BLDR prompts the user for information for developing the threat environment. THREAT_BLDR uses a random number generator to produce x, y coordinates for the individual threats. A random seed, the number of threat categories, and the total threat density are inputs requested. A threat category defines a set of threats with the same radius and cost. The total threat density is the area of the grid that is covered by the threats. In addition to the radius and cost, the user is asked to input the density for each threat category. This defines individual threat category density relating to the total threat density. The sum of the threat category densities is equal to one. After all threats are generated, THREAT_BLDR places an end of file marker in DATAIN to identify the end of input.

The random number generator and portion of code in THREAT_BLDR that creates the random threat environment is the same as that used in [5]. This produces identical threat environments to facilitate comparison between hierarchical planning and heuristic search. The

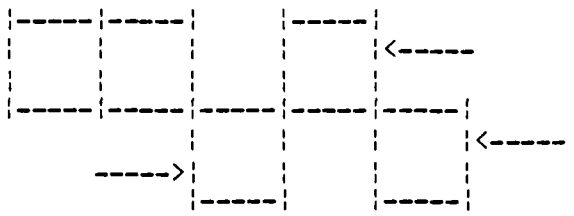


Figure 6.6

Division Lines

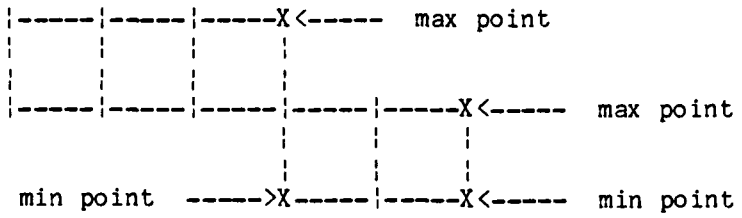


Figure 6.7

Min/Max Points

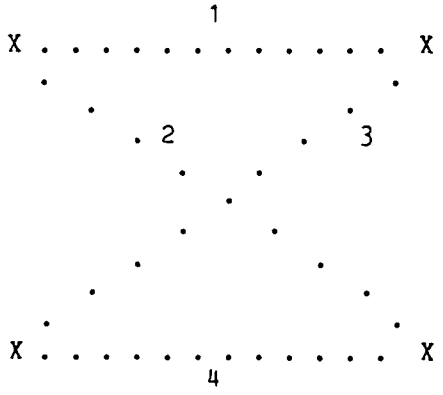


Figure 6.8

Possible Legs

division. The best path to a division is the one with the least cost. If more than one path has the same cost, the shortest path is considered best. This process continues from division to division until the final goal is reached. The best flight path within the block path is then selected.

Cost computation of a flight leg from one point to another involves checking for intersection between a line and a threat circle. If a leg begins outside of a threat and intersects the circle, the threat cost is assigned to the cost of that leg. If a leg begins inside a threat circle, the threat cost is not included in the leg cost because it would have already been assigned to another leg of that path. Legs that are tangent to a threat circle are not considered to be in the threat area and are therefore not assigned the threat cost. A single leg can incur the cost of more than one threat if it intersects several threats. The threat costs are added together to compute the cost of the leg. The total cost for a flight path is the addition of all costs for all the legs in the flight path.

Following the exhaustive search of both block paths, two best flight paths exist, one for each block path. The best flight path is then selected. The best flight path is the path with the lowest cost associated with it. Should both flight paths have the same cost, the shortest flight path is selected.

FLIGHT PATHS

An exhaustive search performed on the block paths produces the best flight path for each block path. The two options that dictate how the exhaustive search is performed are number of divisions and number of points per division. Divisions are sections along a block path that define the beginning and end of a flight leg. The number of divisions will be either 10 or 20 depending on the type of block path. Either 10 or 20 is available for the BIG option and only 20 will be used for the SMALL option. Each division is divided equally based on the number of points per division. Divisions define the distance in the x direction for the legs of a flight path, while the number of points per division define the y coordinates. Divisions have minimum and maximum y coordinates defined. For each division, this is the minimum and maximum points on a vertical line connecting two boxes in the block path. Division lines and their minimum and maximum points are illustrated with a sample block path in Figures 6.6 and 6.7. The number of points per division defines how many points there are on each division line. The minimum number of points is 2 and the maximum is 20. The legs of a flight path start at one division, at each point on that division, and connect to the next division, at each one of its points. For example, if there are two points per division, the number of possible legs to the next division is four. This is illustrated in Figure 6.8. As each leg is generated from one division to the next, cost and length are computed for that flight leg. After all possible legs are generated and their costs computed, from one division to the next, there exists one best path to each point on the next division. For example, if there are five points per division, there will be exactly five best paths to that

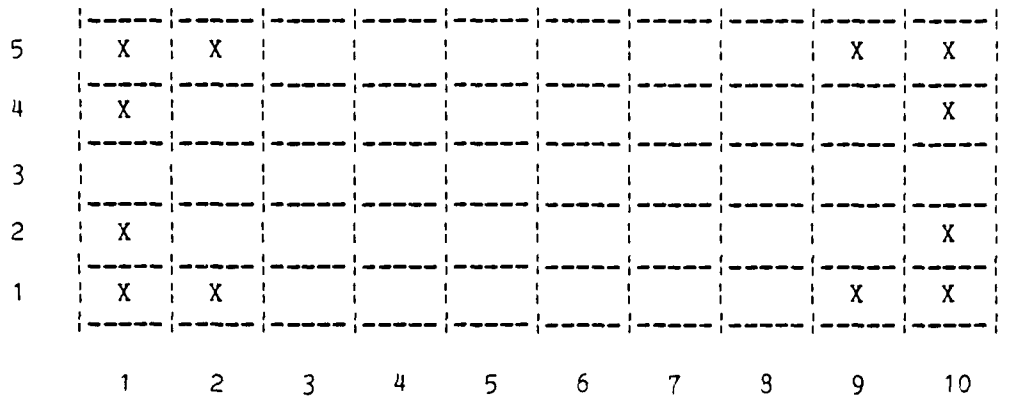


Figure 6.4

Big Block Paths

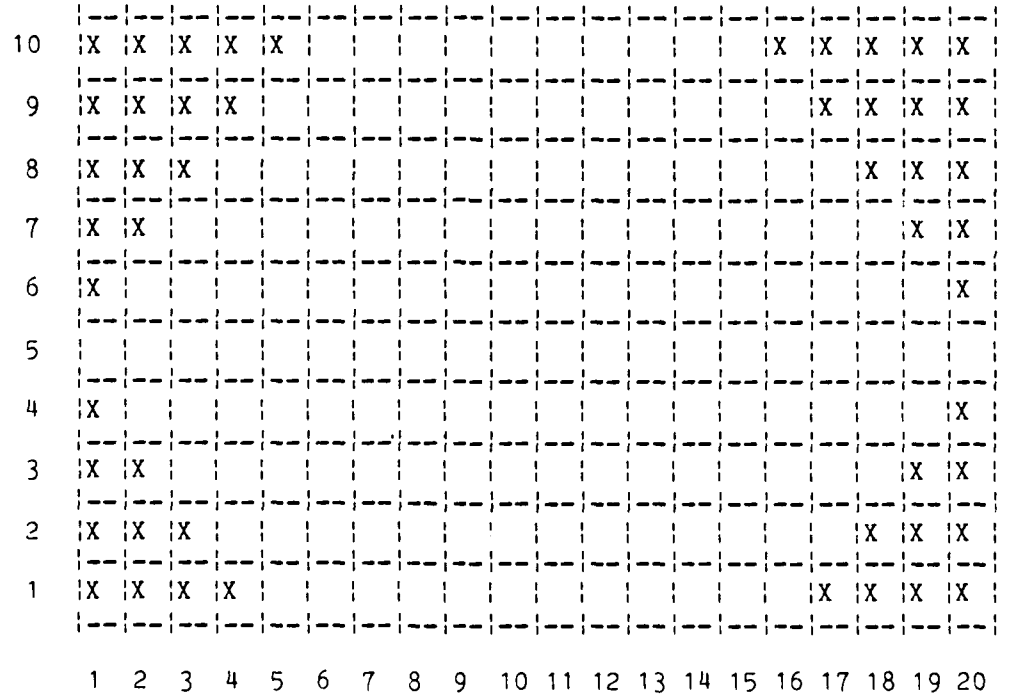


Figure 6.5

Small Block Paths

discussing box values, it was noted that threat values for a box would be stored separately with the DECONFLICT option. If a threat covers two different boxes, then both boxes are assigned the cost of that threat. With the DECONFLICT option, if these two boxes are contained in the same block path, the value from the same threat will only be added to the total score once. This method ensures a threat cost will only be added one time in computing the score for a block path. Both options, ALL or DECONFLICT, are available on every plan except when the block path is BIG and the grid contains small boxes. This is because the small box values are added together to obtain big box values.

After all block paths have been generated and scored, two are selected for an exhaustive search. If there are two block paths that have the best score, then both paths are selected. If more than two block paths are tied with the best score, the two block paths that contain the most boxes exclusive of each other are selected. This is determined by comparing the y box coordinates of each block path with the other block paths that are tied with the best score. If there is only one block path with the best score, it is selected along with a block path that has the next best score. Once two block paths are selected, an exhaustive search is performed on each to produce flight paths.

TOTAL produce the same results. Initially, it was thought that these different options would produce different block paths, therefore, these observations are surprising. However, later results with different threat environments show that these options will at times produce different block paths. The result using heuristic search method produced the same cost, but was a longer path.

THREAT ENVIRONMENT:

RANDOM SEED: 31583
TOTAL THREAT DENSITY. 1.0
NUMBER OF THREATS: 37
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	40	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	17	102	45	132.10
BIG	ALL	TOTAL	10	5	20	17	226	45	122.38
BIG	DECON	TOTAL	10	5	10	17	106	45	132.10
BIG	DECON	TOTAL	10	5	20	17	227	45	122.38
BIG	ALL	TOTAL	20	10	10	17	102	20	105.79
BIG	ALL	TOTAL	20	10	20	17	222	20	108.83
BIG	ALL	CENTER	20	10	10	17	106	20	105.79
BIG	ALL	CENTER	20	10	20	17	225	20	108.83
HEURISTIC SEARCH RESULTS							12	20	113.80

Table 7.1

Results for Test 1

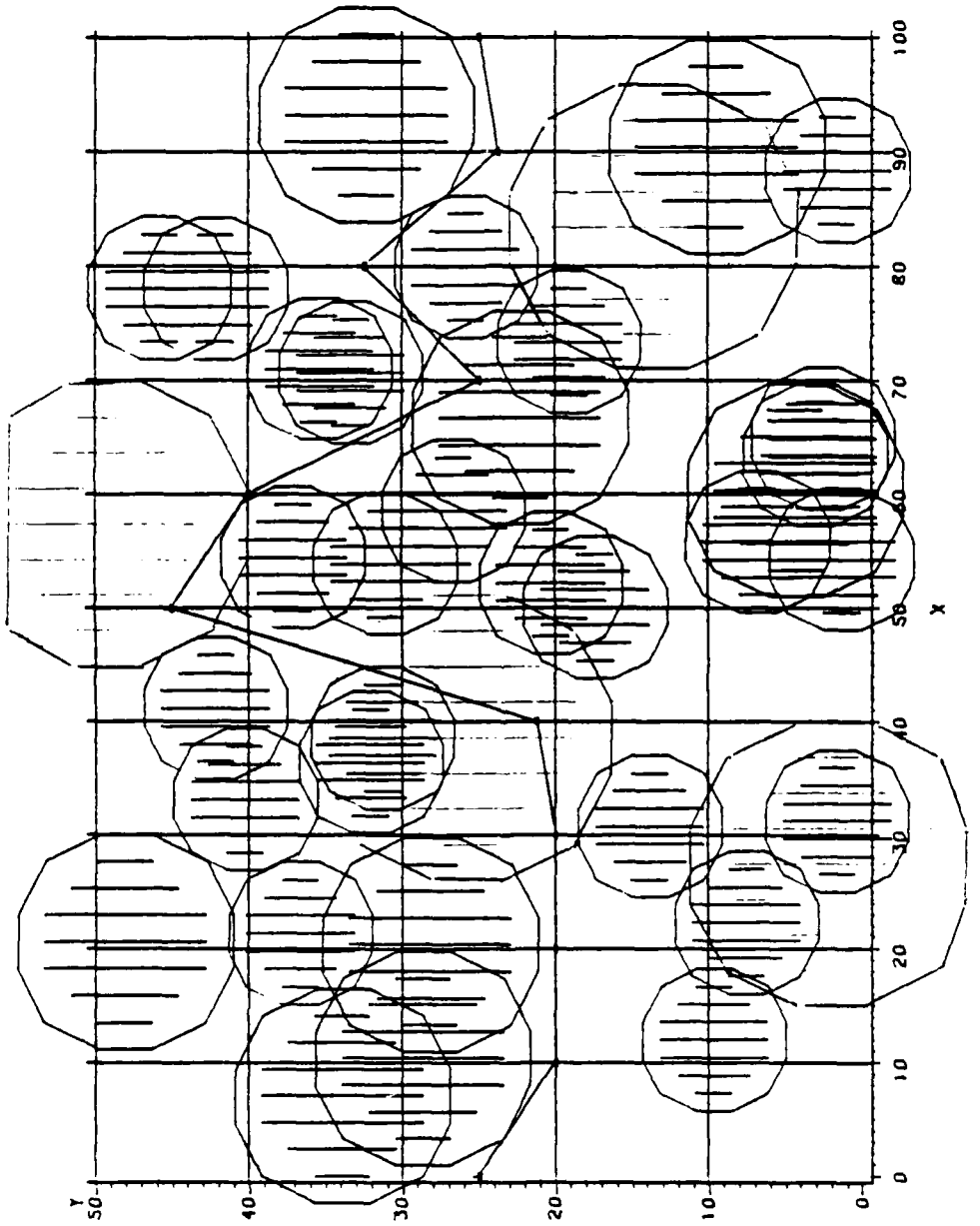


Figure 7.1

Path from Test 1 - 10, 5 Option

Path Cost = 45

Path Length = 132.10

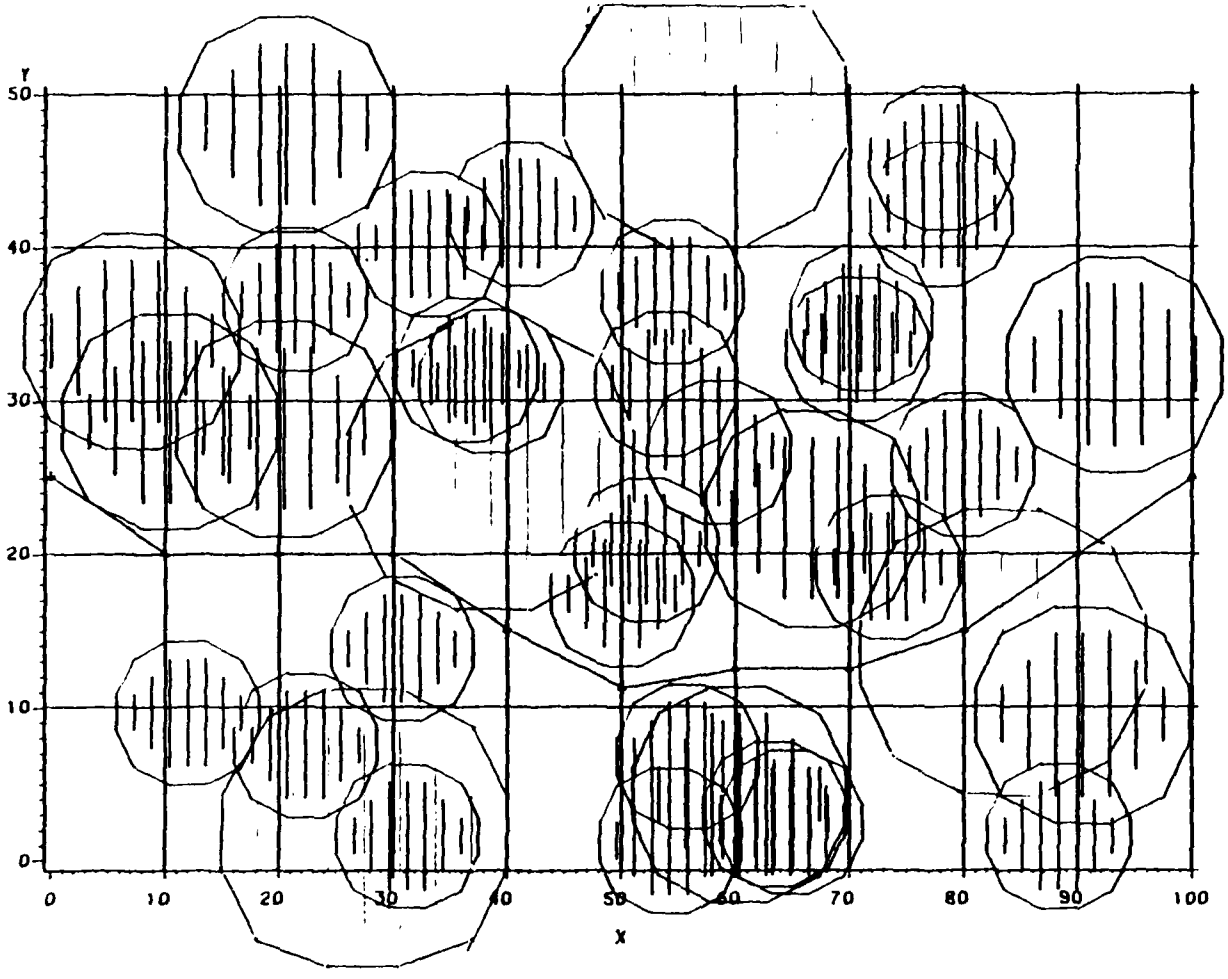


Figure 7.2

Path from Test 1 - 20, 10 Option

Path Cost = 20

Path Length = 105.79

TEST 2

The results for this test show that the x, y values produced the same paths in terms of cost. As with the previous test case, the options ALL/DECONFLICT and CENTER/TOTAL produced similar results. The block paths for the options varied in the first part of the path, but all block paths forced the flight path to reach the goal from the top part of the grid. This produced poor results because the flight path had to pass through at least one of the three high cost threats near the goal. This is illustrated in Figure 7.3. Additionally, all the block paths forced the flight path to intersect the first threat that is closest to the starting point. Therefore, all paths had a cost of 25 assigned after the first leg of the path.

This was the first case in which the type path SMALL option was used. This method produced a block path that moved along the lower section of the grid. Figure 7.4 shows the flight path produced with this option. The path does a good job winding around high cost threats to end up with a path cost of only 20. This method equalled the cost of the path produced by heuristic search. Also, the path using the SMALL option was shorter. The obvious drawback to this method is CPU time. As noted in the beginning of this chapter, the SMALL option is not desirable in terms of CPU time. This was the first test case where the number of points per division made a difference in the cost of the path and not just an improvement in length. With 5 points per division, the SMALL option path had a cost of 45. There were not enough points on a division line for the path to maneuver between threats. When the number of points per division increased to 9 and 17, the path was able to maneuver around and between threats.

THREAT ENVIRONMENT:

RANDOM SEED: 46137
TOTAL THREAT DENSITY: 0.75
NUMBER OF THREATS: 28
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	40	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	17	78	65	109.31
BIG	ALL	TOTAL	10	5	20	17	171	65	112.17
BIG	DECON	TOTAL	10	5	10	17	80	65	109.31
BIG	DECON	TOTAL	10	5	20	17	172	65	112.17
BIG	ALL	TOTAL	20	10	10	17	79	65	109.31
BIG	ALL	TOTAL	20	10	20	17	169	65	111.31
BIG	ALL	CENTER	20	10	10	17	78	65	109.31
BIG	ALL	CENTER	20	10	20	17	175	65	111.31
SMALL	ALL	TOTAL	20	10	20	5	54(HRS)	45	111.18
SMALL	ALL	TOTAL	20	10	20	9	54(HRS)	20	113.50
SMALL	ALL	TOTAL	20	10	20	17	54(HRS)	20	111.88
HEURISTIC SEARCH RESULTS							9185	20	114.96

Table 7.2

Results for Test 2

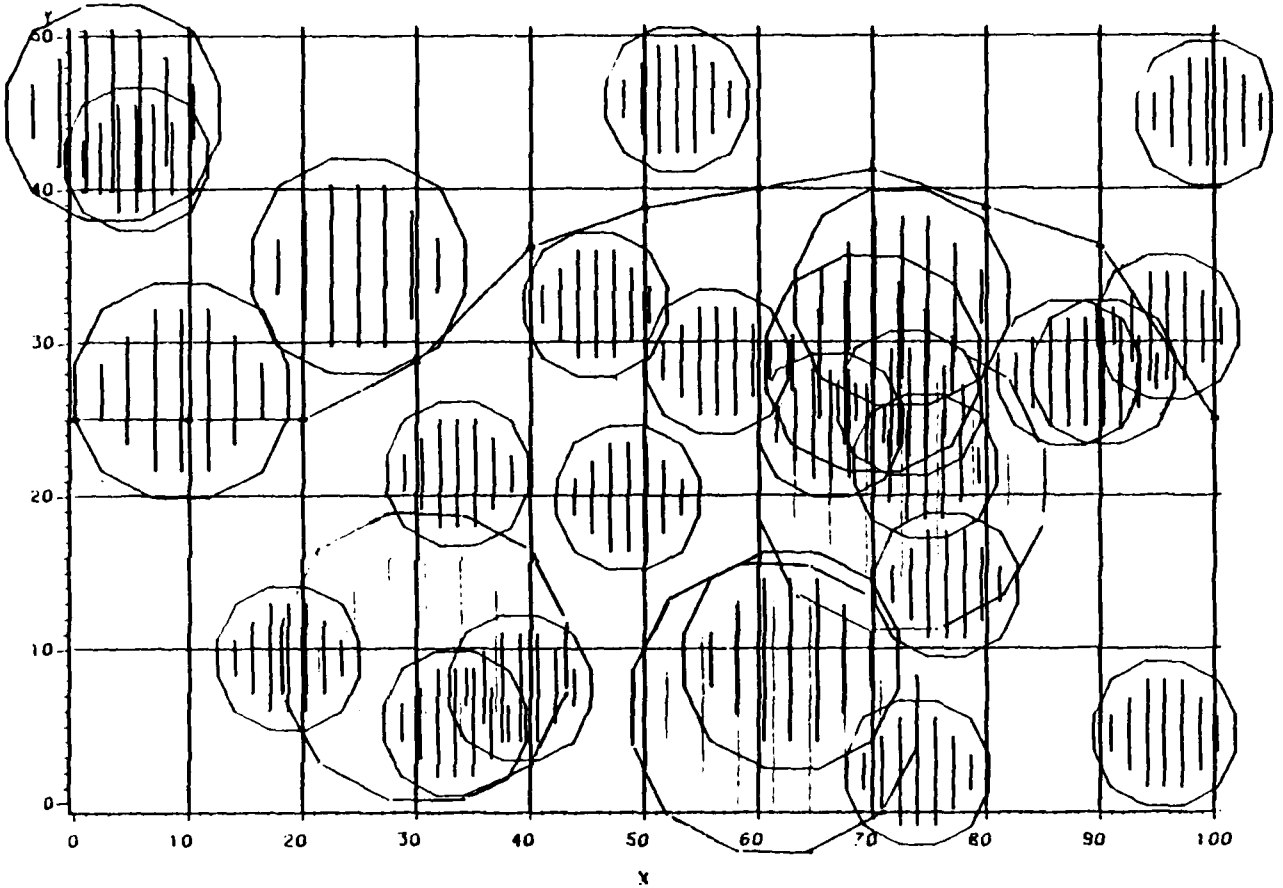


Figure 7.3

Path from Test 2 - BIG Option

Path Cost = 65

Path Length = 109.31

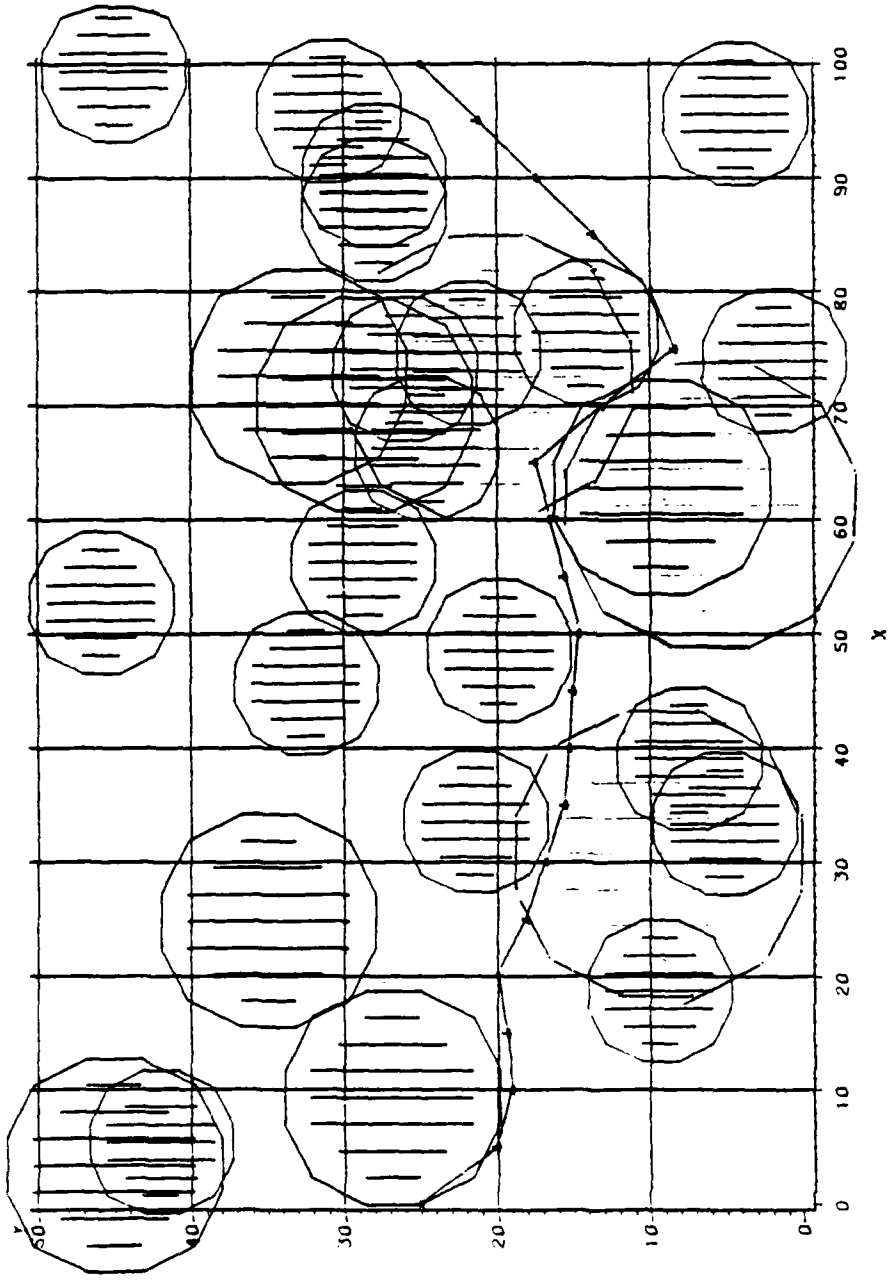


Figure 7.4 Path Cost = 20

Path from Test 2 - SMALL Option Path Length = 111.88

TEST 3

This test case had the most dense threat environment consisting of 75 threats. Using 10, 5 for the x, y values produced better results than 20, 10 this time. All options for 10, 5 resulted in the same block path. The best path produced for this test case is shown in Figure 7.5. From the results of this test case, it was observed the difference the number of divisions make on the path. With x, y values at 20, 10, 10 divisions produced much better results than 20 divisions. This was because the flight path with 10 divisions was along the top border of the grid while the path using 20 divisions intersected threats lower in the grid. This was the only other time that the type path SMALL was used. This option produced a path cost of 230. In this case, small block paths did not result in a better path.

The heuristic search results were better in this test case. The best path hierarchical planning could produce was one that cost 225. The best heuristic search path had a cost of 165. This path maneuvered through the threat environment avoiding many of the high cost threats. To obtain this path, the heuristic search took 1650 seconds of CPU time. In this case, however, the CPU expense is reasonable in order to obtain the path cost of 165. This path is illustrated in Figure 7.6.

THREAT ENVIRONMENT:

RANDOM SEED: 143954
TOTAL THREAT DENSITY: 2.0
NUMBER OF THREATS: 75
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	22	325	107.41
BIG	ALL	TOTAL	10	5	10	9	59	275	106.40
BIG	ALL	TOTAL	10	5	10	17	198	275	105.54
BIG	ALL	TOTAL	10	5	20	5	43	275	110.90
BIG	ALL	TOTAL	10	5	20	9	126	225	107.56
BIG	ALL	TOTAL	10	5	20	17	437	225	106.49
BIG	DECON	TOTAL	10	5	10	5	25	325	107.41
BIG	DECON	TOTAL	10	5	10	9	62	275	106.40
BIG	DECON	TOTAL	10	5	10	17	201	275	105.54
BIG	DECON	TOTAL	10	5	20	5	46	275	110.90
BIG	DECON	TOTAL	10	5	20	9	128	225	107.56
BIG	DECON	TOTAL	10	5	20	17	456	225	106.49
BIG	ALL	TOTAL	20	10	10	5	22	285	121.84
BIG	ALL	TOTAL	20	10	10	9	59	285	116.90
BIG	ALL	TOTAL	20	10	10	17	207	285	115.30
BIG	ALL	TOTAL	20	10	20	5	42	380	114.82
BIG	ALL	TOTAL	20	10	20	9	125	355	110.68
BIG	ALL	TOTAL	20	10	20	17	456	345	116.41
BIG	ALL	CENTER	20	10	10	5	22	285	121.84
BIG	ALL	CENTER	20	10	10	9	59	285	116.90
BIG	ALL	CENTER	20	10	10	17	196	285	115.30
BIG	ALL	CENTER	20	10	20	5	42	380	114.82
BIG	ALL	CENTER	20	10	20	9	126	355	110.68
BIG	ALL	CENTER	20	10	20	17	438	345	116.41
SMALL	ALL	TOTAL	20	10	20	17	54(HRS)	230	112.01
HEURISTIC SEARCH RESULTS							1650	165	112.58

Table 7.3

Results for Test 3

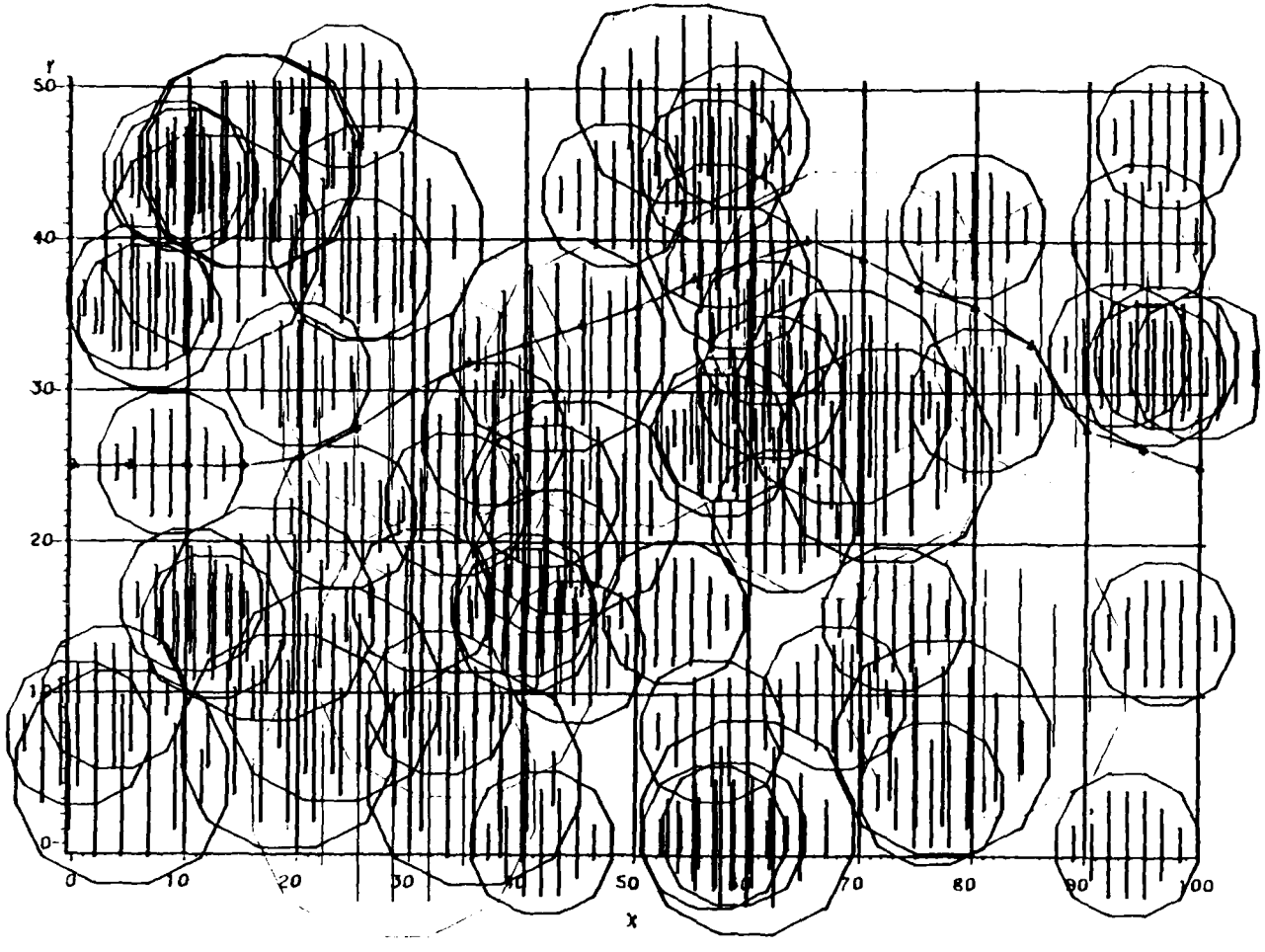


Figure 7.5

Path from Test 3 - 10, 5 Option

Path Cost = 225

Path Length = 106.49

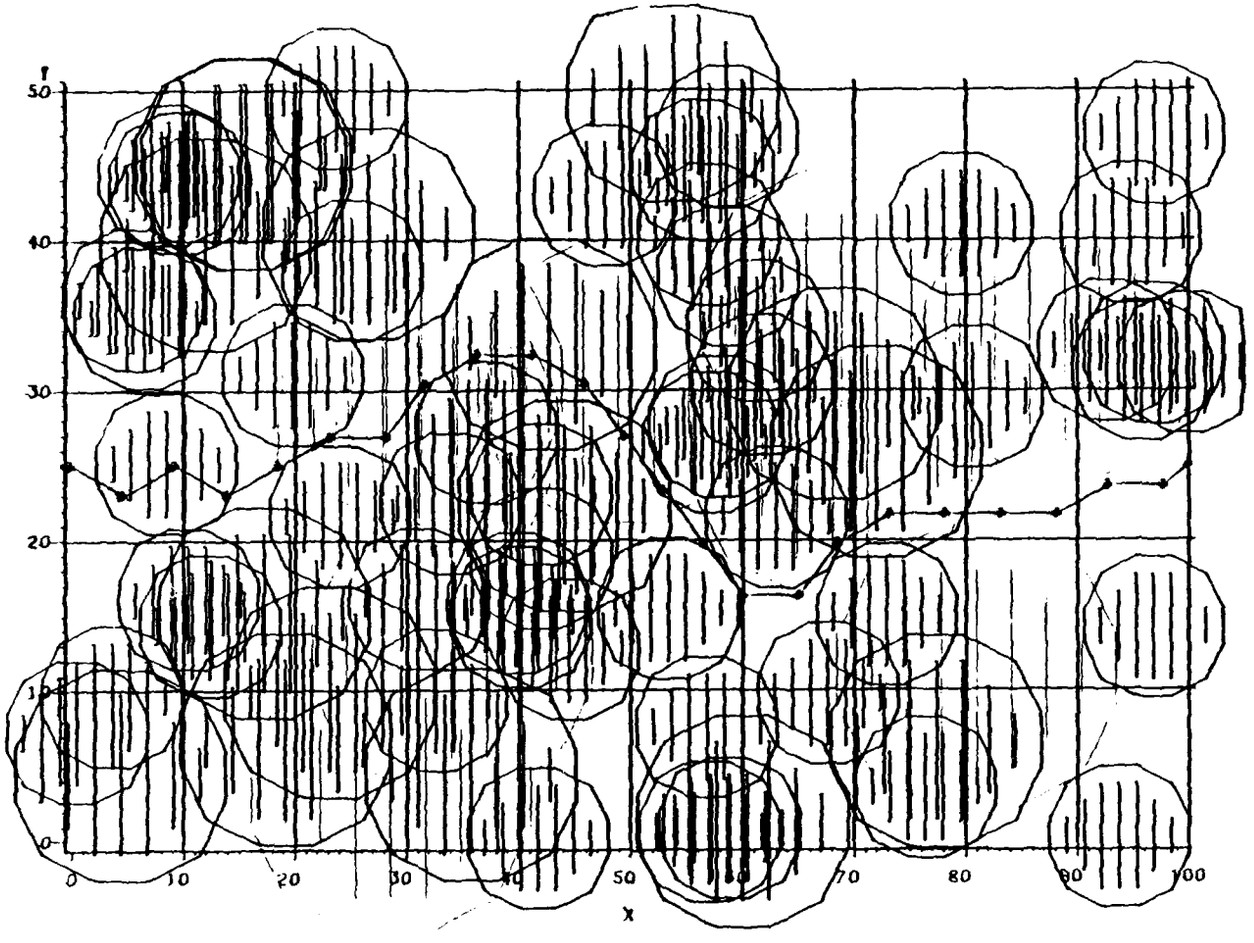


Figure 7.6

Path from Test 3 - Heuristic Search

Path Cost = 165

Path Length = 112.58

TEST 4

This was the first test case in which the DECONFLICT cost option differed from the ALL option. The DECONFLICT option produced much better results than any other method. This was because the block path produced a flight path that could reach the goal from the upper part of the grid rather than from the side or below. This path only intersected one small threat near the goal, while the best path using heuristic search intersected two small threats near the goal because the path came from the bottom part of the grid. The results for the box coverage options TOTAL and CENTER with x, y values 20, 10 were the same again. The best path for this test case was the DECONFLICT cost option using 10 divisions and 17 points per division. This path is shown in Figure 7.7. Figure 7.8 shows the best path using heuristic search.

THREAT ENVIRONMENT:

RANDOM SEED: 21738
TOTAL THREAT DENSITY: 1.5
NUMBER OF THREATS: 57
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	19	220	117.51
BIG	ALL	TOTAL	10	5	10	9	46	205	114.57
BIG	ALL	TOTAL	10	5	10	17	152	170	116.56
BIG	ALL	TOTAL	10	5	20	5	33	280	132.96
BIG	ALL	TOTAL	10	5	20	9	97	280	125.44
BIG	ALL	TOTAL	10	5	20	17	332	230	123.20
BIG	DECON	TOTAL	10	5	10	5	21	205	116.33
BIG	DECON	TOTAL	10	5	10	9	50	205	114.57
BIG	DECON	TOTAL	10	5	10	17	155	155	117.19
BIG	DECON	TOTAL	10	5	20	5	37	205	130.00
BIG	DECON	TOTAL	10	5	20	9	105	205	124.25
BIG	DECON	TOTAL	10	5	20	17	339	155	122.09
BIG	ALL	TOTAL	20	10	10	5	18	255	116.62
BIG	ALL	TOTAL	20	10	10	9	46	230	112.06
BIG	ALL	TOTAL	20	10	10	17	153	230	111.71
BIG	ALL	TOTAL	20	10	20	5	34	230	117.78
BIG	ALL	TOTAL	20	10	20	9	99	230	112.20
BIG	ALL	TOTAL	20	10	20	17	332	205	113.68
BIG	ALL	CENTER	20	10	10	5	18	255	116.62
BIG	ALL	CENTER	20	10	10	9	46	230	112.06
BIG	ALL	CENTER	20	10	10	17	153	230	111.71
BIG	ALL	CENTER	20	10	20	5	33	230	117.78
BIG	ALL	CENTER	20	10	20	9	98	230	112.20
BIG	ALL	CENTER	20	10	20	17	345	205	113.68
HEURISTIC SEARCH RESULTS							4102	210	111.91

Table 7.4

Results for Test 4

TEST 8

The paths from this test either avoid all the threats or intersect one threat with a cost of 50. In this test, the ALL/DECONFLICT cost options have the same results. In both cases, 20 divisions produce the better results. The TOTAL/CENTER box coverage options produce the same results and again 20 divisions is better than 10. This is another example where the number of divisions is significant. In all options, 20 divisions resulted in a path cost of 0, whereas the cost was 50 when the number of divisions is 10. This is because with 10 divisions, the path is unable to avoid a threat near the goal. When the number of divisions is 20, the path is able to set itself up for a move between the two threats. With 20 divisions, the flight legs are shorter and therefore the path is able to maneuver more. The best path for this test is illustrated in Figure 7.14. Heuristic search was also able to avoid all the threats, however, the path length was slightly longer.

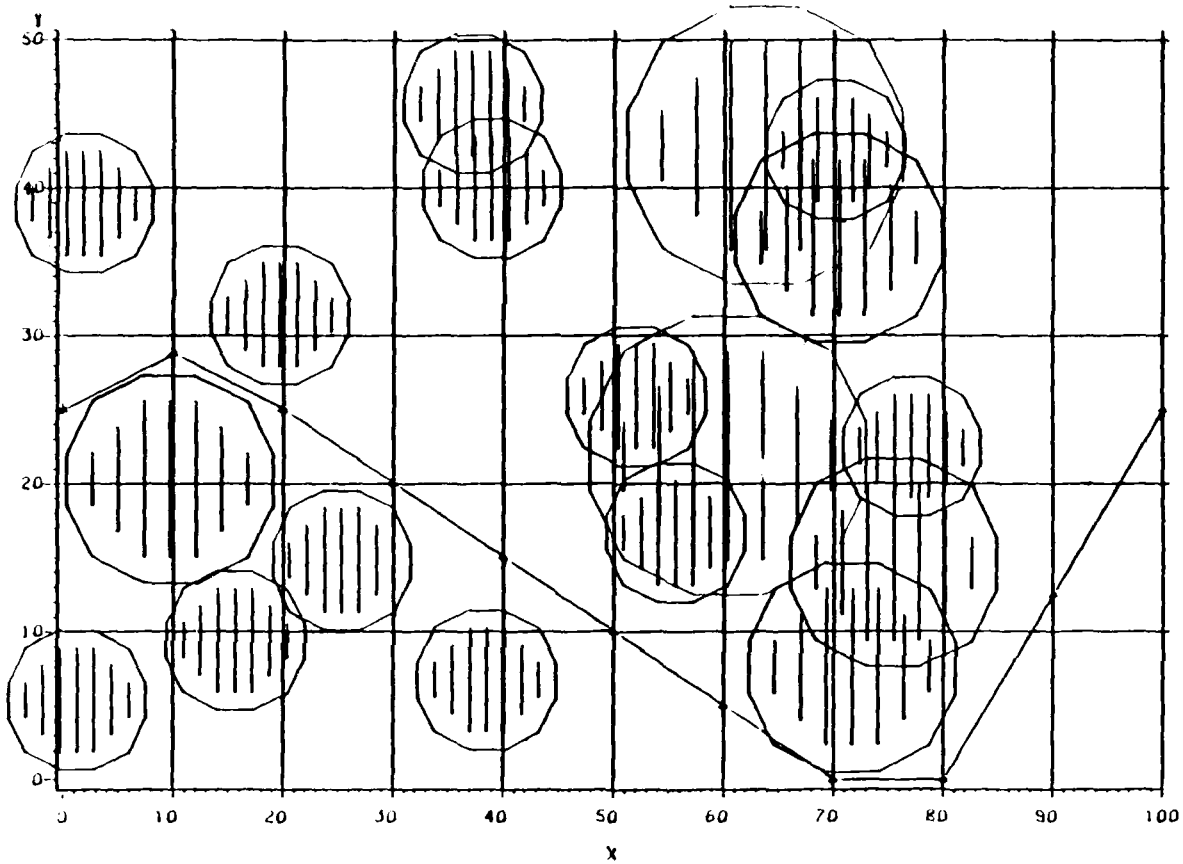


Figure 7.13

Path from Test 7 - DECONFLICT Option

Path Cost = 0

Path Length = 119.28

THREAT ENVIRONMENT:

RANDOM SEED: 28547
TOTAL THREAT DENSITY: 0.5
NUMBER OF THREATS: 18
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	9	10	118.94
BIG	ALL	TOTAL	10	5	10	9	18	10	118.62
BIG	ALL	TOTAL	10	5	10	17	54	10	117.55
BIG	ALL	TOTAL	10	5	20	5	14	50	117.40
BIG	ALL	TOTAL	10	5	20	9	36	50	111.57
BIG	ALL	TOTAL	10	5	20	17	117	50	110.07
BIG	DECON	TOTAL	10	5	10	5	11	0	120.43
BIG	DECON	TOTAL	10	5	10	9	20	0	119.28
BIG	DECON	TOTAL	10	5	10	17	55	0	119.28
BIG	DECON	TOTAL	10	5	20	5	16	0	132.07
BIG	DECON	TOTAL	10	5	20	9	37	0	126.22
BIG	DECON	TOTAL	10	5	20	17	116	0	124.08
BIG	ALL	TOTAL	20	10	10	5	9	10	116.29
BIG	ALL	TOTAL	20	10	10	9	18	10	114.01
BIG	ALL	TOTAL	20	10	10	17	54	10	112.87
BIG	ALL	TOTAL	20	10	20	5	14	10	123.10
BIG	ALL	TOTAL	20	10	20	9	37	10	120.75
BIG	ALL	TOTAL	20	10	20	17	117	10	119.45
BIG	ALL	CENTER	20	10	10	5	9	10	111.84
BIG	ALL	CENTER	20	10	10	9	19	10	105.07
BIG	ALL	CENTER	20	10	10	17	54	10	104.76
BIG	ALL	CENTER	20	10	20	5	14	10	114.19
BIG	ALL	CENTER	20	10	20	9	36	10	108.14
BIG	ALL	CENTER	20	10	20	17	121	10	106.20
HEURISTIC SEARCH RESULTS							3	10	104.57

Table 7.7

Results for Test 7

TEST 7

There were six different block paths produced by the options in this test case. Block paths in this test covered virtually all possible ways of avoiding the threats. This is due to the small threat density that produced only 18 threats. Therefore, there are many ways for block paths to maneuver around threats. Some went to the top of the grid, while others stayed in the middle or at the bottom. The best block path was with x, y at 10, 5 and the cost option equal to DECONFLICT. This block path produced a flight path that avoided all threats in the grid to give a cost of 0. The best the heuristic search could produce was a path with a cost of 10. Again, the results for the cost options ALL and DECONFLICT were different, with DECONFLICT having better results. The option with x, y at 20, 10 also produced different results with the CENTER/TOTAL box coverage options. This time the CENTER option produced a shorter flight path. The best flight path for this test case is given in Figure 7.13. This path runs on the x-axis for 10 units, thereby just missing a threat. The y coordinate for the threat center just referred to is 7.61. Since the radius of this threat is 7.5, that leaves just 0.11 units for the flight path to avoid the threat. The path in Figure 7.13 accomplishes this.

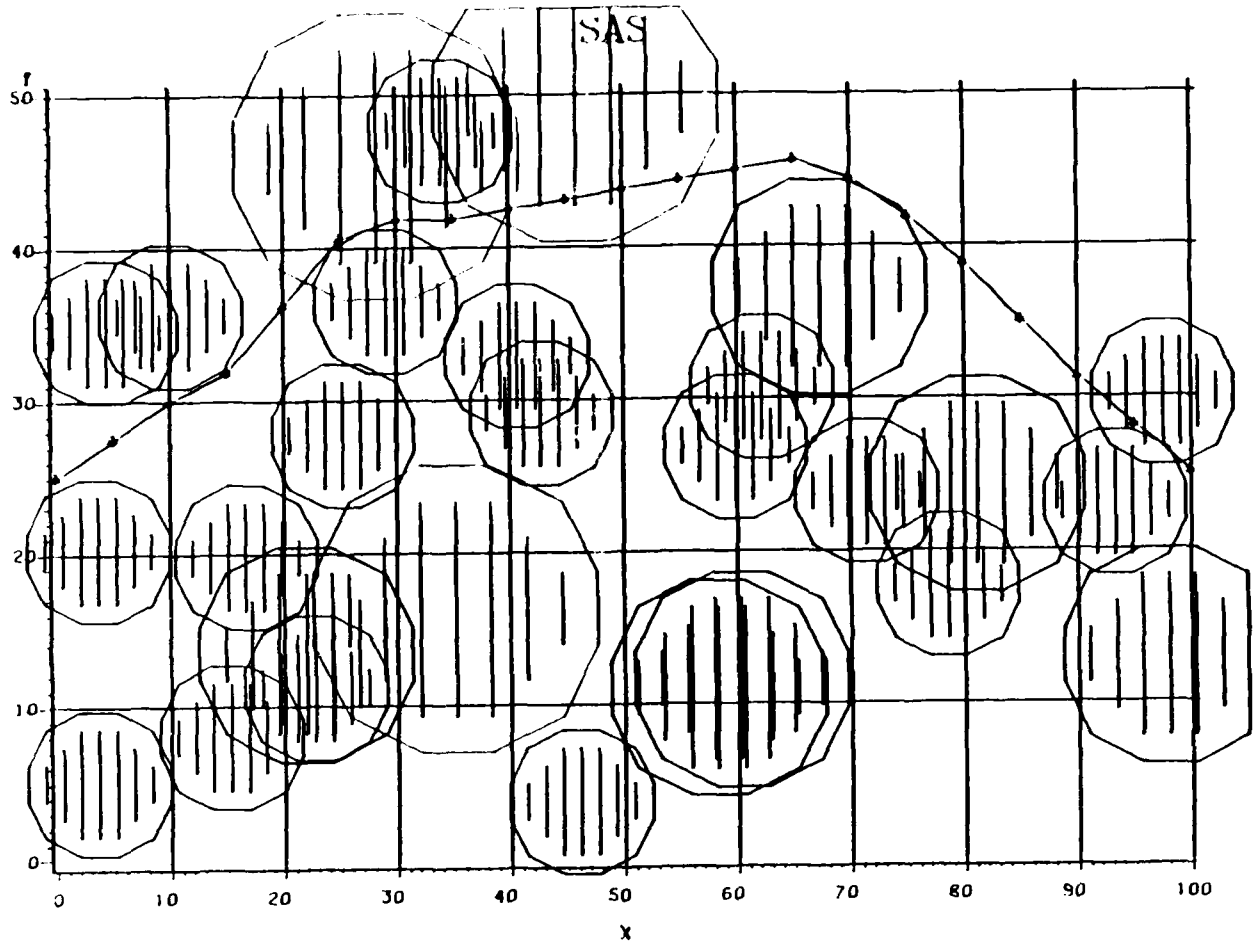


Figure 7.12

Path from Test 6 - 20 Divisions

Path Cost = 70

Path Length = 111.13

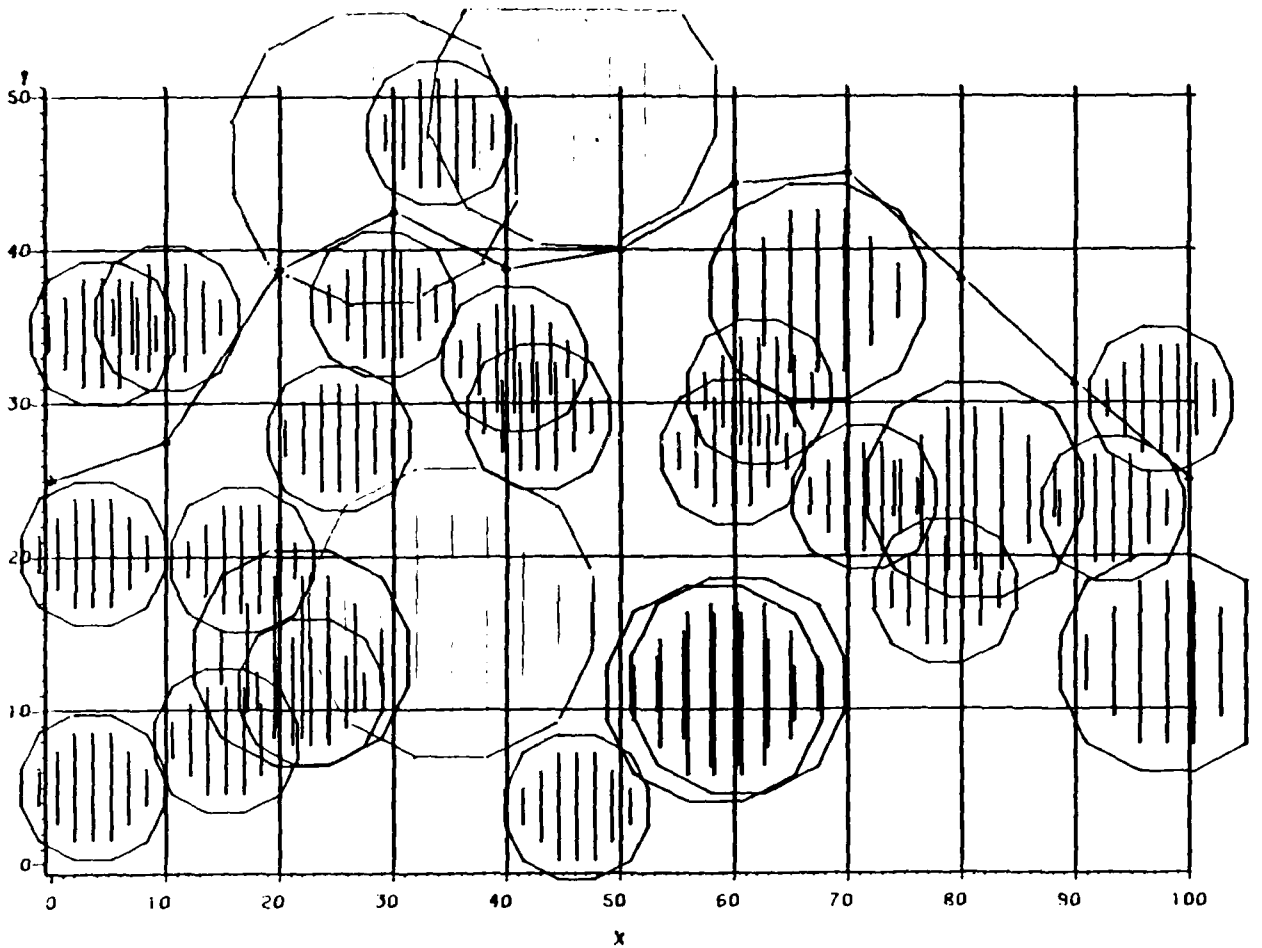


Figure 7.11

Path from Test 6 - TOTAL Option

Path Cost = 60

Path Length = 113.80

THREAT ENVIRONMENT:

RANDOM SEED: 120775
TOTAL THREAT DENSITY: 0.75
NUMBER OF THREATS: 28
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	11	110	111.88
BIG	ALL	TOTAL	10	5	10	9	26	70	125.65
BIG	ALL	TOTAL	10	5	10	17	79	70	122.89
BIG	ALL	TOTAL	10	5	20	5	19	95	121.26
BIG	ALL	TOTAL	10	5	20	9	54	60	117.63
BIG	ALL	TOTAL	10	5	20	17	173	60	116.62
BIG	DECON	TOTAL	10	5	10	5	14	110	111.88
BIG	DECON	TOTAL	10	5	10	9	30	70	125.65
BIG	DECON	TOTAL	10	5	10	17	83	70	122.89
BIG	DECON	TOTAL	10	5	20	5	22	95	121.26
BIG	DECON	TOTAL	10	5	20	9	55	60	117.63
BIG	DECON	TOTAL	10	5	20	17	174	60	116.62
BIG	ALL	TOTAL	20	10	10	5	11	110	112.44
BIG	ALL	TOTAL	20	10	10	9	25	110	109.68
BIG	ALL	TOTAL	20	10	10	17	78	60	113.80
BIG	ALL	TOTAL	20	10	20	5	20	120	113.01
BIG	ALL	TOTAL	20	10	20	9	51	70	113.13
BIG	ALL	TOTAL	20	10	20	17	171	70	112.25
BIG	ALL	CENTER	20	10	10	5	11	110	112.44
BIG	ALL	CENTER	20	10	10	9	26	70	113.72
BIG	ALL	CENTER	20	10	10	17	79	60	113.82
BIG	ALL	CENTER	20	10	20	5	19	70	114.36
BIG	ALL	CENTER	20	10	20	9	52	70	112.23
BIG	ALL	CENTER	20	10	20	17	172	70	111.13
HEURISTIC SEARCH RESULTS							810	60	106.82

Table 7.6

Results for Test 6

TEST 6

This was the first test case where the box coverage options CENTER and TOTAL produced different results. With x, y values equal to 20, 10, the TOTAL option resulted in the best path. This is shown in Figure 7.11. In both CENTER and TOTAL options, the best path resulted when the number of divisions was 10. When the number of divisions was 20, the flight paths intersected an additional threat with a cost of 10. The best path with 20 divisions with x, y at 20, 10 is shown in Figure 7.12. The heuristic search method also found a path with a cost of 60, but it was shorter than any of the above paths.

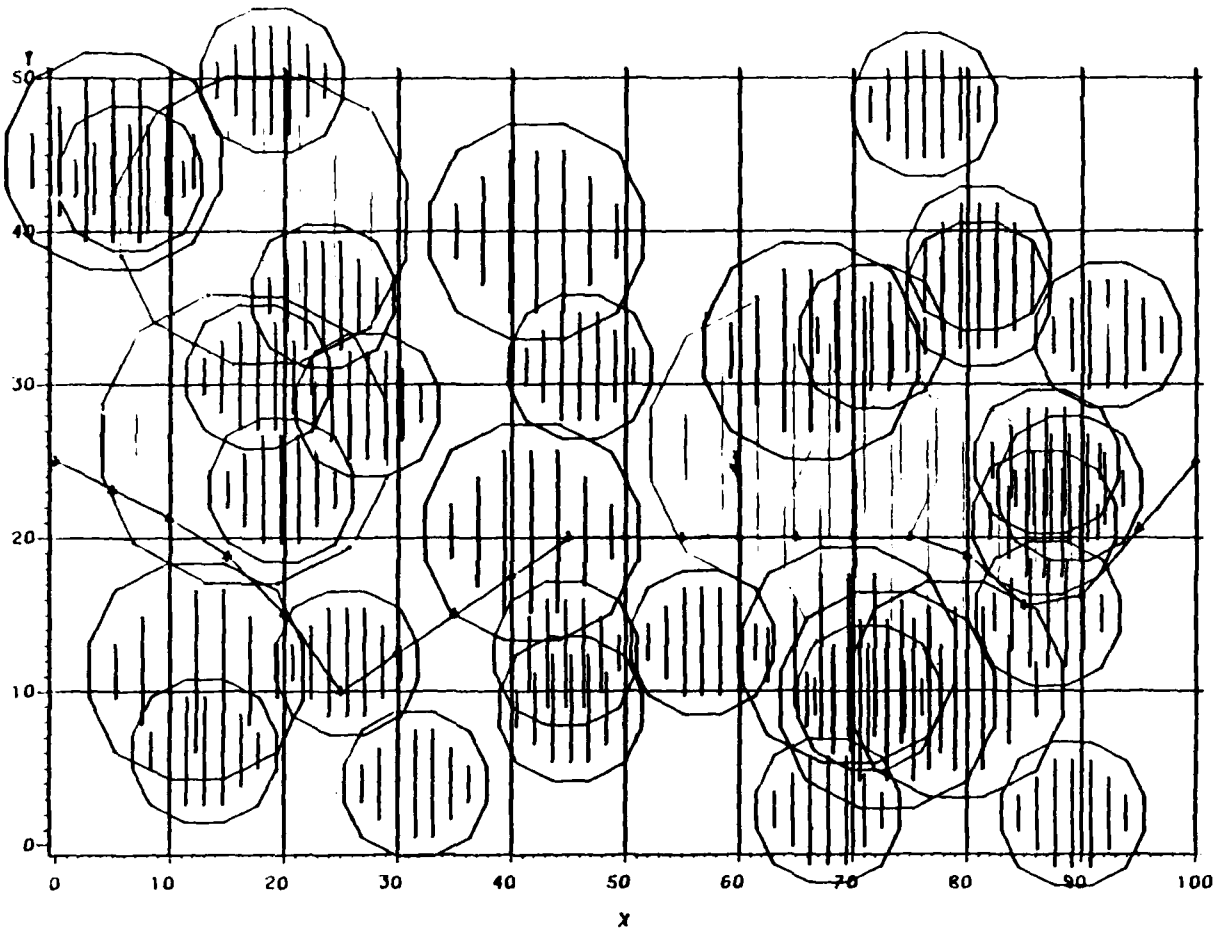


Figure 7.10

Path from Test 5 - 20 Divisions

Path Cost = 155

Path Length = 111.33

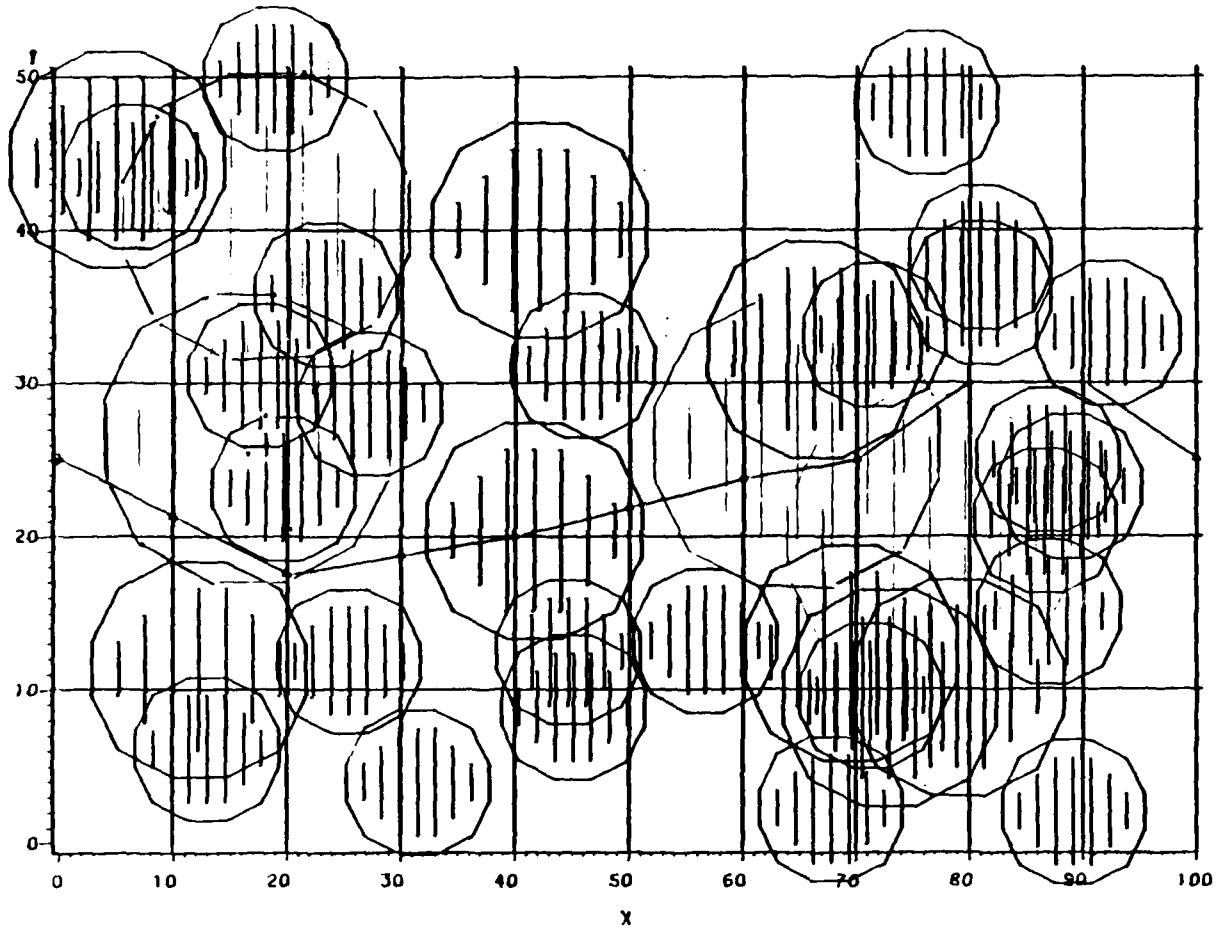


Figure 7.9

Path from Test 5 - 10 Divisions

Path Cost = 105

Path Length = 104.30

THREAT ENVIRONMENT:

RANDOM SEED: 89204
TOTAL THREAT DENSITY: 1.0
NUMBER OF THREATS: 37
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	13	105	108.86
BIG	ALL	TOTAL	10	5	10	9	32	105	104.62
BIG	ALL	TOTAL	10	5	10	17	103	105	104.30
BIG	ALL	TOTAL	10	5	20	5	24	155	114.79
BIG	ALL	TOTAL	10	5	20	9	65	155	113.23
BIG	ALL	TOTAL	10	5	20	17	228	155	111.33
BIG	DECON	TOTAL	10	5	10	5	16	105	119.32
BIG	DECON	TOTAL	10	5	10	9	35	105	115.99
BIG	DECON	TOTAL	10	5	10	17	105	105	114.96
BIG	DECON	TOTAL	10	5	20	5	28	105	121.46
BIG	DECON	TOTAL	10	5	20	9	69	105	119.54
BIG	DECON	TOTAL	10	5	20	17	228	105	116.01
BIG	ALL	TOTAL	20	10	10	5	13	110	116.33
BIG	ALL	TOTAL	20	10	10	9	32	110	114.33
BIG	ALL	TOTAL	20	10	10	17	101	110	113.61
BIG	ALL	TOTAL	20	10	20	5	24	160	113.30
BIG	ALL	TOTAL	20	10	20	9	67	110	114.23
BIG	ALL	TOTAL	20	10	20	17	220	110	113.67
BIG	ALL	CENTER	20	10	10	5	13	110	116.33
BIG	ALL	CENTER	20	10	10	9	32	110	114.33
BIG	ALL	CENTER	20	10	10	17	100	110	113.61
BIG	ALL	CENTER	20	10	20	5	23	160	113.30
BIG	ALL	CENTER	20	10	20	9	65	110	114.23
BIG	ALL	CENTER	20	10	20	17	220	110	113.67
HEURISTIC SEARCH RESULTS							1933	105	106.82

Table 7.5

Results for Test 5

TEST 5

The results using x, y values at 10, 5 were better than 20, 10. Also with 10, 5, the ALL and DECONFLICT cost options were different. However, where DECONFLICT was better in test 4, the ALL option was better this time. The block path for the ALL option produced the shortest flight path with the least cost of 105. In the ALL option with x, y at 10, 5, the number of divisions had a significant effect on the path cost in this test. Even though the block paths were the same, the number of divisions forced the flight path to be different. The difference was that when the number of divisions was 20, the path intersected one more threat than it did with 10 divisions. The additional threat contacted had a cost of 50. Both these paths are shown in Figures 7.9 and 7.10. The cost for a path using heuristic search was also 105, however, the hierarchical planning method produced a shorter flight path. The paths from both methods are similar and intersect the same threats.

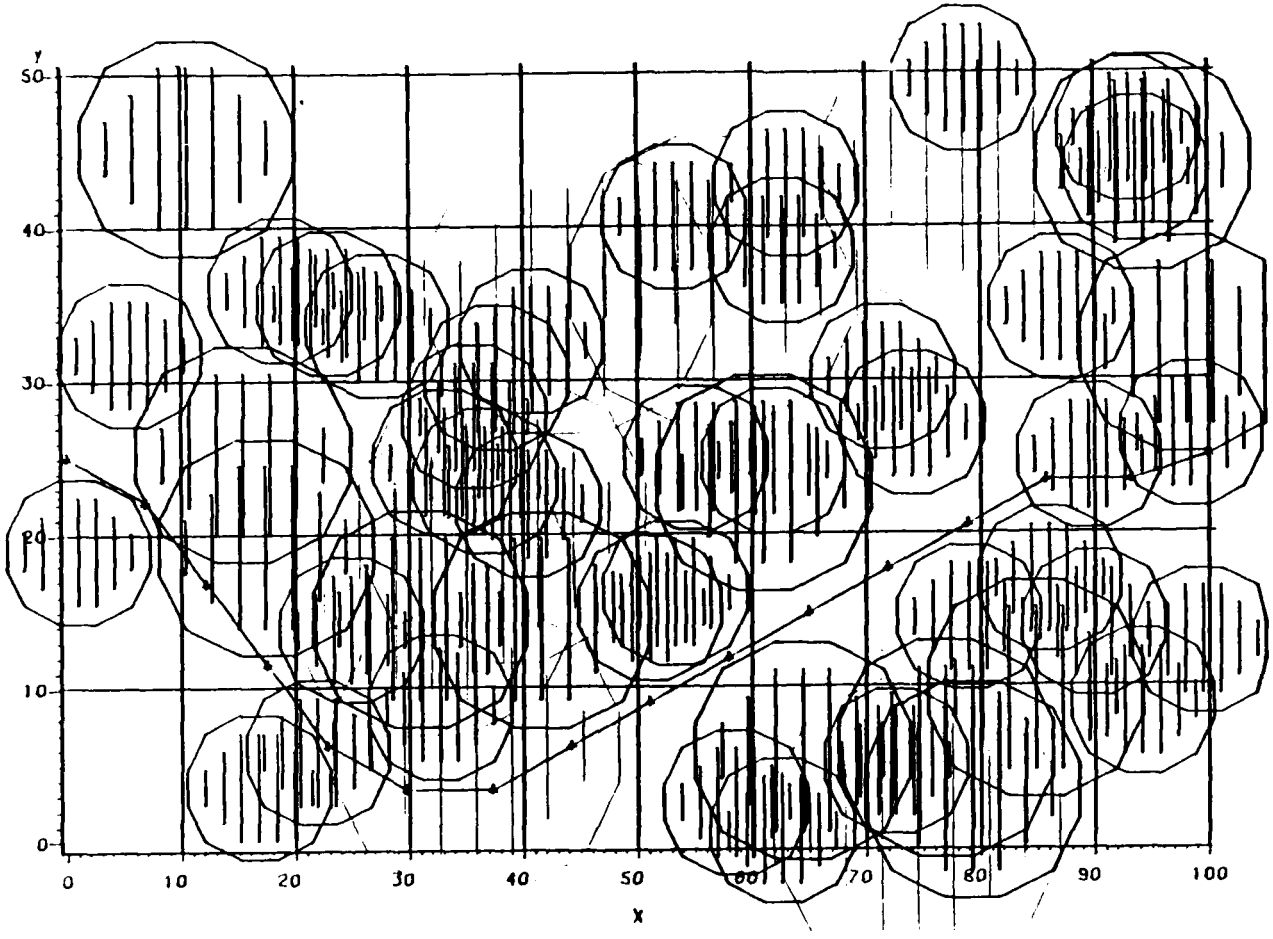


Figure 7.8

Path from Test 4 - Heuristic Search

Path Cost = 210

Path Length = 111.91

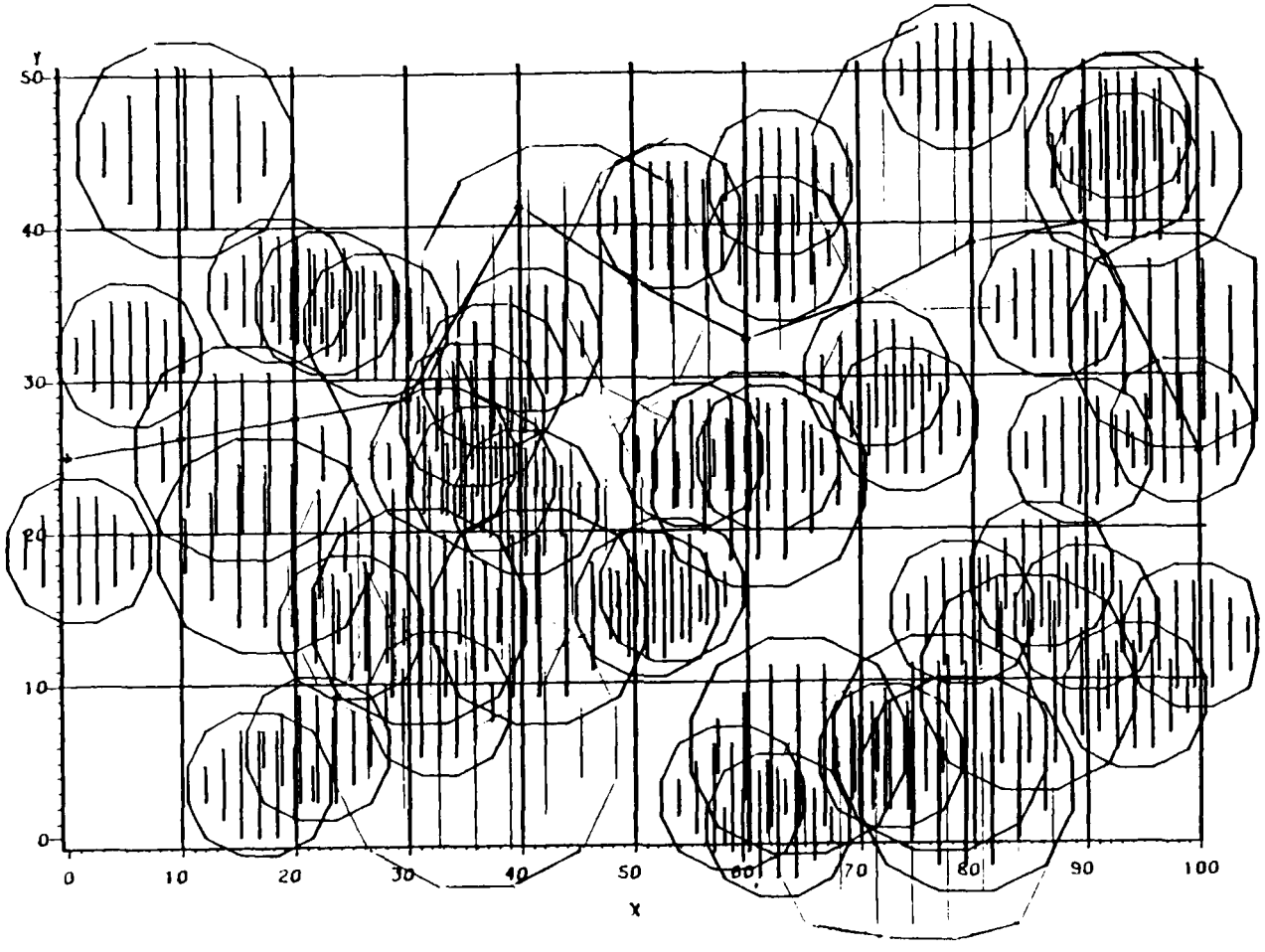


Figure 7.7

Path from Test 4 - DECONFLICT Option

Path Cost = 155

Path Length = 117.19

THREAT ENVIRONMENT:

RANDOM SEED: 39306
TOTAL THREAT DENSITY: 0.5
NUMBER OF THREATS: 18
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	9	50	108.86
BIG	ALL	TOTAL	10	5	10	9	18	50	108.54
BIG	ALL	TOTAL	10	5	10	17	53	50	107.74
BIG	ALL	TOTAL	10	5	20	5	14	0	111.83
BIG	ALL	TOTAL	10	5	20	9	35	0	109.49
BIG	ALL	TOTAL	10	5	20	17	114	0	108.56
BIG	DECON	TOTAL	10	5	10	5	11	50	108.86
BIG	DECON	TOTAL	10	5	10	9	21	50	108.54
BIG	DECON	TOTAL	10	5	10	17	55	50	107.74
BIG	DECON	TOTAL	10	5	20	5	17	0	111.83
BIG	DECON	TOTAL	10	5	20	9	38	0	109.49
BIG	DECON	TOTAL	10	5	20	17	116	0	108.56
BIG	ALL	TOTAL	20	10	10	5	9	50	108.86
BIG	ALL	TOTAL	20	10	10	9	19	50	108.54
BIG	ALL	TOTAL	20	10	10	17	54	50	107.91
BIG	ALL	TOTAL	20	10	20	5	14	0	111.83
BIG	ALL	TOTAL	20	10	20	9	35	0	109.49
BIG	ALL	TOTAL	20	10	20	17	115	0	108.94
BIG	ALL	CENTER	20	10	10	5	9	50	108.86
BIG	ALL	CENTER	20	10	10	9	18	50	108.54
BIG	ALL	CENTER	20	10	10	17	53	50	107.91
BIG	ALL	CENTER	20	10	20	5	14	0	111.83
BIG	ALL	CENTER	20	10	20	9	37	0	109.49
BIG	ALL	CENTER	20	10	20	17	113	0	108.94
HEURISTIC SEARCH RESULTS							54	0	109.49

Table 7.8

Results for Test 8

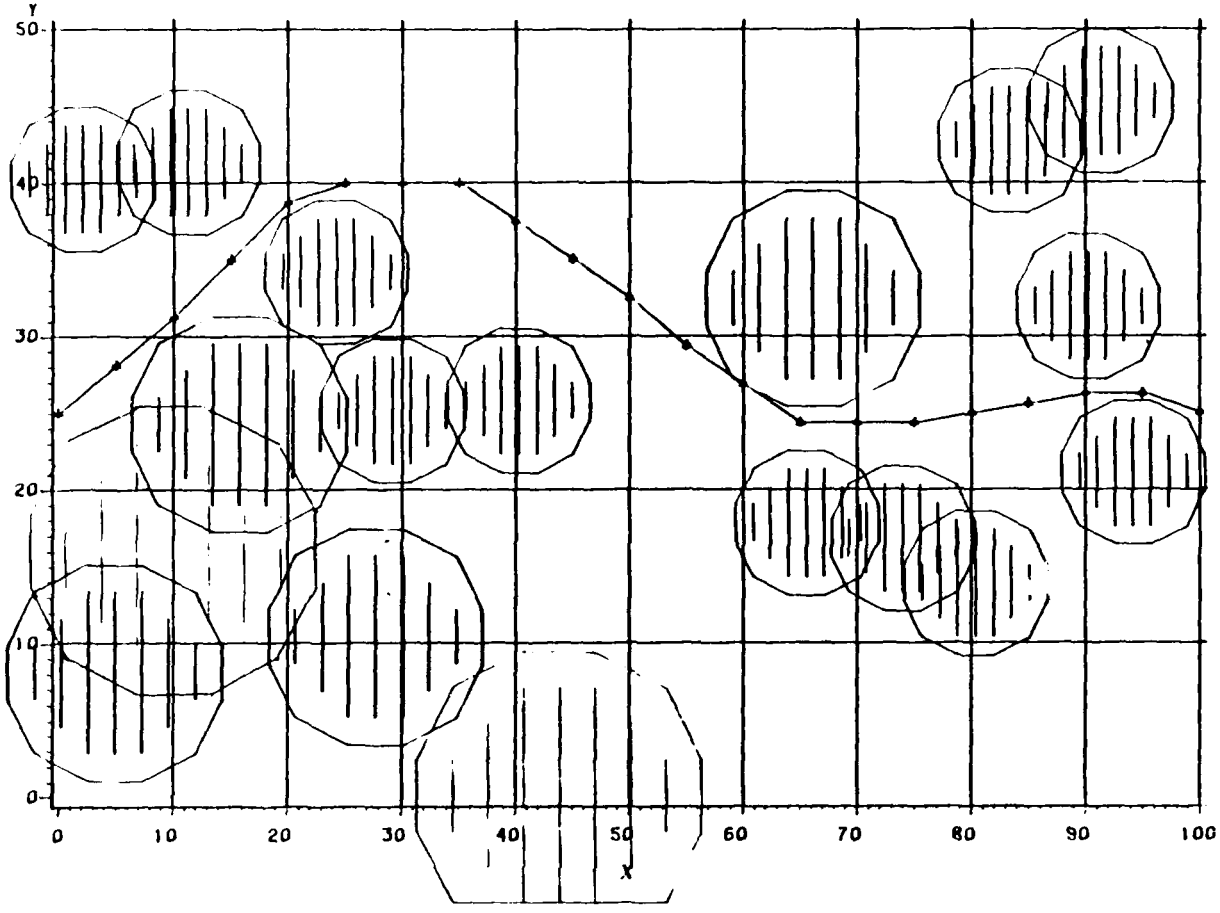


Figure 7.14

Path from Test 8 - 20 Divisions

Path Cost = 0

Path Length = 108.56

CONCLUSIONS

The result of using hierarchical planning to generate flight paths through a random threat environment produced better paths than did heuristic search in five of the eight test cases. If the results using the SMALL option with undesirable CPU time is included, the better paths were produced in six cases. This reinforces statements made earlier that planning increases problem-solving power. Several conclusions can be drawn from the test results. First, when using big block paths to solve the problem, it is better to evaluate the threat environment based on how the threats effect big boxes (x, y values at 10, 5). This method produced better results in six out of the eight tests. It would seem that evaluating the threats based on small boxes would provide a better representation of the threat environment. However, adding the small boxes together to process a big block path in some cases distorts actual threat concentration. The cost options ALL/DECONFLICT were different in three test cases. In two of the three cases, the DECONFLICT option produced a better path in terms of cost. This is because the DECONFLICT option only counts a threat cost once in determining the best block path. Intuitively, the DECONFLICT method of cost computation makes more sense. The box coverage options CENTER and TOTAL were equally effective. The number of divisions used in all the options sometimes made significant differences in path cost. As noted in test 3, 20 divisions will allow a path to maneuver around and between threats. However, 10 divisions will sometimes produce a flight path that will

miss threats altogether, while 20 divisions in that same block path may force the flight path to intersect a threat as was the case in test 5. The determining factor between using 10 or 20 divisions is the location of threats contained in a block path.

HPLAN implementation has some shortcomings. The quality of the final solution depends on block paths. An exhaustive search through a block path will find the best flight path within that block path. However, there is no guarantee that the block path chosen will contain the best flight path for the problem. With some threat environments, there may be more than two best block paths. Since only two are selected for exhaustive search, the best possible path may be contained in a block path not selected. Therefore, that portion of HPLAN that searches through a block path is admissable (a search algorithm that is guaranteed to find an optimal path to a goal, if one exists). However, the HPLAN algorithm as a whole is not admissable. The solution may be to use exhaustive search for all the best block paths. In some cases, this will be almost as expensive as an exhaustive search of the entire problem space. Another problem with HPLAN exists with the type path SMALL option. As discussed previously, the CPU time involved is undesirable. If CPU time is not a consideration, then the SMALL option may produce better paths.

Based on the results in this thesis, the hierarchical planning method was an excellent technique to use for solving this type of problem. The use of heuristics in [5] also has merit. Perhaps a combination of both methods would produce still better results. One possibility would be to use an 'intelligent' heuristic to choose the best block path and then perform an exhaustive search. Another approach

would be to select more than two block paths and then apply heuristics to search through the block paths for the best path. In any case, use of one or more artificial intelligence techniques for problem-solving is recommended.

APPENDIX A

```
(* program hplan generates aircraft routes and finds a path through *)
(* random hostile environments using the artificial intelligence    *)
(* technique of hierarchical planning                               *)

program hplan(input, output, datain, pathout);

    const
                x_limit = 100;
        max_xboxes = 20;
        max_yboxes = 10;
        max_num_threats = 100;
        max_threats_per_box = 10;
        max_pts_on_line = 20;

    type
        block_size = (BIG, SMALL);
        cost_option = (ALL, DECONFLICT);
        coverage = (CENTER, TOTAL);
        threat_array = array[1..max_num_threats] of real;
        cost_array = array[1..max_num_threats] of integer;
        box_array = array[1..max_threats_per_box] of integer;
        grid_array = array[1..max_xboxes, 1..max_yboxes] of integer;
        threat_add_array = array[1..max_num_threats] of integer;
        path_array = array[1..max_xboxes] of integer;
        flight_path_array = array[1..max_pts_on_line, 1..max_xboxes] of real;
        final_flight_array = array[1..max_xboxes] of real;

        threat_values = record
                cx : threat_array;
                cy : threat_array;
                value : cost_array;
                radius : threat_array;
```

```
        end;

next_point = record
        xcoor : flight_path_array;
        ycoor : flight_path_array;
        end;

flight_paths = record
        xpt : flight_path_array;
        ypt : flight_path_array;
        conflict : flight_path_array;
        len : flight_path_array;
        end;

threat_rec = record
        num : integer;
        value : box_array;
        kind : box_array;
        end;

box_threat = array[1..max_xboxes, 1..max_yboxes] of threat_rec;

threat_add = record
        add : threat_add_array;
        x : threat_add_array;
        y : threat_add_array;
        end;

block_rec = record
        x : path_array;
        y : path_array;
        score : integer;
        end;

block_path = array[1..max_xboxes] of block_rec;
```

```
        flight_rec = record
                x : final_flight_array;
                y : final_flight_array;
                cost : real;
                len : real;
                end;

        final_flight = array[1..max_xboxes] of flight_rec;

    var
        threat_in : threat_values;
        threat, visit : grid_array;
        cur_xpath, cur_ypath, best_xpath, best_ypath,
          next_best_xpath, next_best_ypath : path_array;
        next_pt : next_point;
        path : flight_paths;
        threat_decon : box_threat;
        threat_added : threat_add;
        datain, pathout : text;
        type_path : block_size;
        block : block_path;
        flight : final_flight;
        option : cost_option;
        box_coverage : coverage;
        x_max, y_max, tot_cpu_time, threat_num,
          num_div, pts_on_line, path_num : integer;
        first_xpt, first_ypt, last_xpt, last_ypt : real;

(* distance computes the distance between two points *)

function distance(x1, y1, x2, y2 : real) : real;

    begin
    distance := sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2);
    end;
```

```
(* initialize intializes data structures for processing *)

procedure initialize;

    var
            i, j, k : integer;
    begin
    reset(datain);
    rewrite(pathout);
    threat_num := 0;
    path_num := 0;
    first_xpt := 0.0;
    first_ypt := 25.0;
    last_xpt := 100.0;
    last_ypt := 25.0;
    for i:= 1 to max_xboxes do
            for j := 1 to max_yboxes do
                    threat[i, j] := 0;
    for i := 1 to max_xboxes do
            begin
            for j := 1 to max_yboxes do
                    begin
                    threat_decon[i, j].num := 0;
                    for k := 1 to max_threats_per_box do
                            begin
                            threat_decon[i, j].value[k] := 0;
                            threat_decon[i, j].kind[k] := 0;
                            end;
                    end;
            end;
    for i := 1 to max_num_threats do
            begin
            threat_added.add[i] := 0;
            threat_added.x[i] := 0;
            threat_added.y[i] := 0;
            end;
```

```
    end;

(* print_inputs prints the program options selected *)
(* and the threat environment                       *)

procedure print_inputs;
    var
            i : integer;
    begin
    writeln(pathout, 'HPLAN RESULTS:');
    writeln(pathout);
    writeln(pathout);
    writeln(pathout, 'OPTIONS SELECTED:');
    writeln(pathout);
    writeln(pathout, '   TYPE OF BLOCK PATHS -- ', type_path);
    writeln(pathout, '   BLOCK PATH COSTS    -- ', option);
    writeln(pathout, '   BOX THREAT COVERAGE -- ', box_coverage);
    writeln(pathout, '   XY BLOCK LIMITS     -- ', x_max : 3, y_max : 3);
    writeln(pathout, '   NUMBER OF DIVISIONS -- ', num_div : 3);
    writeln(pathout, '   POINTS PER DIVISION -- ', pts_on_line : 3);
    writeln(pathout);
    writeln(pathout);
    writeln(pathout, ' INPUT THREAT ENVIRONMENT:');
    writeln(pathout);
    writeln(pathout, '   TOTAL NUMBER OF THREATS -- ', threat_num : 1);
    writeln(pathout);
    writeln(pathout, '   THREAT INPUTS -- X Y RADIUS COST');
    writeln(pathout);
    for i := 1 to threat_num do
            writeln(pathout, '      ', threat_in.cx[i] : 8:2,
                    threat_in.cy[i] : 8:2, threat_in.radius[i] : 6:1,
                    threat_in.value[i] : 5);
    page(pathout);
    end;

(* read_data reads input data from file datain built by threat_bldr *)
```

```
procedure read_data;

    var
            x_in, y_in, t_rad : real;
            t_val : integer;
    begin
    readln(datain, type_path);
    readln(datain, option);
    readln(datain, box_coverage);
    readln(datain, x_max, y_max);
    readln(datain, num_div, pts_on_line);
    readln(datain, x_in, y_in, t_val, t_rad);
    while (x_in <> -1) do
            (* read and store all threats *)
            begin
            threat_num := threat_num + 1;
            threat_in.cx[threat_num] := x_in;
            threat_in.cy[threat_num] := y_in;
            threat_in.value[threat_num] := t_val;
            threat_in.radius[threat_num] := t_rad;
            readln(datain, x_in, y_in, t_val, t_rad);
            end;
    print_inputs;
    end;

(* compute_block_coor takes the center of a threat and returns the *)
(* xy box coordinate where the center of the threat is located     *)

procedure compute_block_coor(x_in, y_in : real; var x, y : integer);

    var
            x_rnd, y_rnd, remand, units : integer;
    begin
    units := x_limit div x_max;
    x_rnd := round(x_in);
```

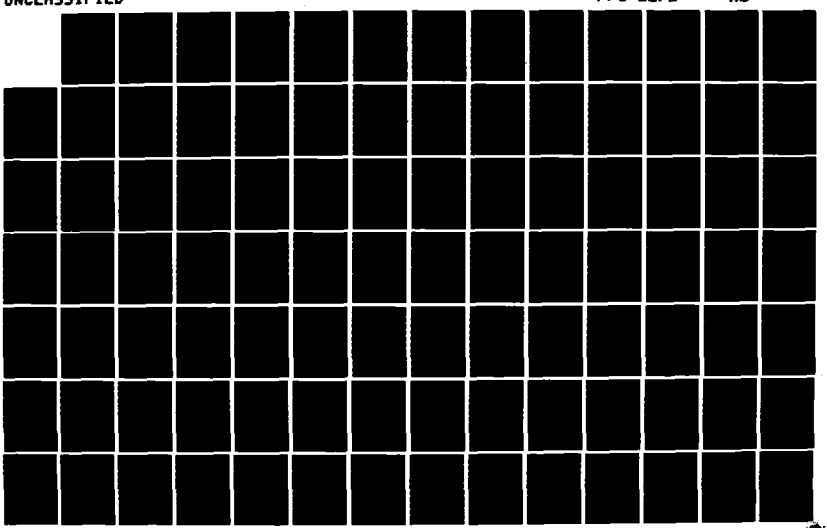
```
    y_rnd := round(y_in);
    if x_rnd < 1 then
            x_rnd := 1;
     f y_rnd < 1 then
            y_rnd := 1;
    remand := x_rnd mod units;
    x := x_rnd div units;
    if remand <> 0 then
            x := x + 1;
    remand := y_rnd mod units;
    y := y_rnd div units;
    if remand <> 0 then
            y := y + 1;
    end;

(* check_box calculates the distance between the center of a box *)
(* and the center of a threat, and if the distance is less than  *)
(* or equal to the threat radius, the threat value is assigned   *)
(* to the box                                                    *)

procedure check_box(x, y, inx : integer);

    var
            i, units : integer;
            half_units, x_mid, y_mid, dist : real;
    begin
    units := x_limit div x_max;
    half_units := units / 2;
    x_mid := x * units - half_units;  (* calculate the xy coordinates *)
    y_mid := y * units - half_units;  (* for the center of a box      *)
    dist := distance(threat_in.cx[inx], threat_in.cy[inx], x_mid, y_mid);
    if dist <= threat_in.radius[inx] then
            (* threat is contained in the box *)
            begin
            if option = DECONFLICT then
                    begin
```

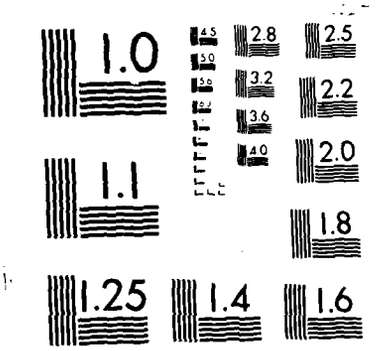
```
                    i := threat_decon[x, y].num + 1;
                    threat_decon[x, y].num := i;
                    threat_decon[x, y].value[i] := threat_in.value[inx];
                    threat_decon[x, y].kind[i] := inx;
                    end
            else
                    threat[x, y] := threat[x, y] + threat_in.value[inx];
            end;
    end;

(* threat_analysis takes the xy box coordinate containing a threat *)
(* and checks adjacent boxes for containment in the threat radius  *)

procedure threat_analysis(x, y, threat_inx : integer);

    var
            small_boxes : boolean;
    begin
    if (x_max = 20) and (box_coverage = TOTAL) then
            small_boxes := true
    else
            small_boxes := false;
    (* check boxes above *)
    if (y + 1) <= y_max then
            begin
            check_box(x, y + 1, threat_inx);
            if (small_boxes) and (y + 2 <= y_max) then
                    check_box(x, y + 2, threat_inx);
            end;
    (* check boxes below *)
    if (y - 1) >= 1 then
            begin
            check_box(x, y - 1, threat_inx);
            if (small_boxes) and (y - 2 >= 1) then
                    check_box(x, y - 2, threat_inx);
            end;
```

```
(* check boxes to the left *)
if (x - 1) >= 1 then
        begin
        check_box(x - 1, y, threat_inx);
        if (small_boxes) and (x - 2 >= 1) then
                check_box(x - 2, y, threat_inx);
        end;
(* check boxes to the right *)
if (x + 1) <= x_max then
        begin
        check_box(x + 1, y, threat_inx);
        if (small_boxes) and (x + 2 <= x_max) then
                check_box(x + 2, y, threat_inx);
        end;
(* check boxes in the upper left *)
if ((x - 1 >= 1) and (y + 1 <= y_max)) then
        begin
        check_box(x - 1, y + 1, threat_inx);
        if (small_boxes) then
                begin
                if (x - 2 >= 1) and (y + 1 <= y_max) then
                        check_box(x - 2, y + 1, threat_inx);
                if (x - 1 >= 1) and (y + 2 <= y_max) then
                        check_box(x - 1, y + 2, threat_inx);
                if (x - 2 >= 1) and (y + 2 <= y_max) then
                        check_box(x - 2, y + 2, threat_inx);
                end;
        end;
(* check boxes in the upper right *)
if ((x + 1 <= x_max) and (y + 1 <= y_max)) then
        begin
        check_box(x + 1, y + 1, threat_inx);
        if (small_boxes) then
                begin
                if (x + 2 <= x_max) and (y + 1 <= y_max) then
                        check_box(x + 2, y + 1, threat_inx);
```

AD-A156 905 GENERATION OF FLIGHT PATHS USING HIERARCHICAL PLANNING 2/3
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
K B KLINE 1985 AFIT/CI/NR-85-37T

UNCLASSIFIED F/G 12/2 NL





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

```
                if (x + 1 <= x_max) and (y + 2 <= y_max) then
                        check_box(x + 1, y + 2, threat_inx);
                if (x + 2 <= x_max) and (y + 2 <= y_max) then
                        check_box(x + 2, y + 2, threat_inx);
                end;
        end;
(* check boxes in the lower left *)
if ((x - 1 >= 1) and (y - 1 >= 1)) then
        begin
        check_box(x - 1, y - 1, threat_inx);
        if (small_boxes) then
                begin
                if (x - 2 >= 1) and (y - 1 >= 1) then
                        check_box(x - 2, y - 1, threat_inx);
                if (x - 1 >= 1) and (y - 2 >= 1) then
                        check_box(x - 1, y - 2, threat_inx);
                if (x - 2 >= 1) and (y - 2 >= 1) then
                        check_box(x - 2, y - 2, threat_inx);
                end;
        end;
(* check boxes in the lower right *)
if ((x + 1 <= x_max) and (y - 1 >= 1)) then
        begin
        check_box(x + 1, y - 1, threat_inx);
        if (small_boxes) then
                begin
                if (x + 2 <= x_max) and (y - 1 >= 1) then
                        check_box(x + 2, y - 1, threat_inx);
                if (x + 1 <= x_max) and (y - 2 >= 1) then
                        check_box(x + 1, y - 2, threat_inx);
                if (x + 2 <= x_max) and (y - 2 >= 1) then
                        check_box(x + 2, y - 2, threat_inx);
                end;
        end;
end;
```

```
(* print_threats prints a matrix of the grid with the threat values *)
(* that are assigned to the boxes                                   *)

procedure print_threats;

    var
            x, y, i : integer;
    begin
    writeln(pathout);
    writeln(pathout, 'BOX THREAT VALUES');
    writeln(pathout);
    if option = DECONFLICT then
            begin
            for y := y_max downto 1 do
                    begin
                    write(pathout, y : 3);
                    for i := 1 to max_threats_per_box do
                            begin
                            for x := 1 to x_max do
                                    write(pathout, threat_decon[x,y].value[i] : 3, '    ');
                            writeln(pathout);
                            if i <> max_threats_per_box then
                                    write(pathout, '    ');
                            end;
                    end;
            end
    else
            begin
            for y := y_max downto 1 do
                    begin
                    write(pathout, y : 3);
                    for x := 1 to x_max do
                            write(pathout, threat[x, y] : 3, '    ');
                    writeln(pathout);
                    end;
            end;
```

```
    for x := 1 to x_max do
            write(pathout, '    ', x : 3);
    writeln(pathout);
    end;

(* add_small_boxes adds the values in small boxes to obtain values for *)
(* big boxes.  Each big box contains four small boxes                   *)

procedure add_small_boxes;

    var
            i, j, x, y : integer;
            sm_threat : grid_array;
    begin
    (* store values in array sm_threat *)
    for i := 1 to x_max do
            for j := 1 to y_max do
                    sm_threat[i, j] := threat[i, j];
    j := 0;
    (* reset grid from small to big boxes *)
    x_max := 10;
    y_max := 5;
    for x := 1 to x_max do
            begin
            i := 0;
            for y := 1 to y_max do
                    begin
                    (* add four small boxes to get big box values *)
                    threat[x, y] := sm_threat[x + j, y + i] +
                                    sm_threat[x + j, y + i + 1] +
                                    sm_threat[x + j + 1, y + i] +
                                    sm_threat[x + j + 1, y + i + 1];
                    i := i + 1;
                    end;
            j := j + 1;
            end;
```

```
    end;

(* process_threats evaluates the threat environment and places *)
(* threat values in the boxes                                  *)

procedure process_threats;

    var
            i, j, x, y : integer;
    begin
    tot_cpu_time := clock;  (* system clock *)
    for i := 1 to threat_num do
            (* process each threat *)
            begin
            compute_block_coor(threat_in.cx[i], threat_in.cy[i], x, y);
            if option = DECONFLICT then
                    (* store threat value in box *)
                    begin
                    j := threat_decon[x, y].num + 1;
                    threat_decon[x, y].num := j;
                    threat_decon[x, y].value[j] := threat_in.value[i];
                    threat_decon[x, y].kind[j] := i
                    end
            else (* add in threat to box value *)
                    threat[x, y] := threat[x, y] + threat_in.value[i];
            threat_analysis(x, y, i);
            end;
    print_threats;
    if (type_path = BIG) and (x_max = 20) then
            (* add up small box values to get big box values *)
            begin
            add_small_boxes;
            print_threats;
            end;
    end;
```

```
(* block_paths computes and stores the best block paths *)

procedure block_paths;

    const
            max_ties = 10;

    type
            ties = record
                    x : path_array;
                    y : path_array;
                    end;

            tie_array = array[1..max_ties] of ties;

    var
            x, y, inx, cur_score, best_score,
              next_best_score, num_ties : integer;
            tie_paths : tie_array;

(* add_to_list adds a box to the current block path *)

procedure add_to_list(x, y : integer);

    begin
    inx := inx + 1;
    cur_xpath[inx] := x;
    cur_ypath[inx] := y;
    end;

(* init_block_paths initializes the structures needed *)
(* for block path computation                         *)

procedure init_block_paths;

    var
```

```
        i, j : integer;
begin
x := 1;
if type_path = SMALL then
        y := 5
else
        y := 3;
inx := 0;
num_ties := 0;
add_to_list(x, y);
(* initialize beginning score for block paths *)
if option = DECONFLICT then
        begin
        cur_score := 0;
        if threat_decon[x, y].num <>0 then
                begin
                for i := 1 to threat_decon[x, y].num do
                        begin
                        cur_score := cur_score + threat_decon[x, y].value[i];
                        j := threat_decon[x, y].kind[i];
                        threat_added.add[j] := 1;
                        threat_added.x[j] := x;
                        threat_added.y[j] := y;
                        end;
                end;
        end
else
        cur_score := threat[x, y];
best_score := 100000;
next_best_score := 100001;
(* initialize all boxes to not visited yet *)
for i := 1 to x_max do
        for j := 1 to y_max do
                visit[i, j] := 0;
visit[x, y] := 1;
(* initialize array to hold the best block path *)
```

```
for i := 1 to x_max do
        begin
        best_xpath[i] := 0;
        best_ypath[i] := 0;
        end;
(* mark all boxes in grid that cannot possibly *)
(* lead to the goal as not used                *)
if option = DECONFLICT then
        begin
        if (x_max = 10) then
                begin
                threat_decon[9, 1].value[1] := -1;
                threat_decon[9, 5].value[1] := -1;
                threat_decon[10, 1].value[1] := -1;
                threat_decon[10, 2].value[1] := -1;
                threat_decon[10, 4].value[1] := -1;
                threat_decon[10, 5].value[1] := -1;
                end
        else
                begin
                threat_decon[16, 10].value[1] := -1;
                threat_decon[17, 1].value[1] := -1;
                threat_decon[17, 9].value[1] := -1;
                threat_decon[17, 10].value[1] := -1;
                threat_decon[18, 1].value[1] := -1;
                threat_decon[18, 2].value[1] := -1;
                threat_decon[18, 8].value[1] := -1;
                threat_decon[18, 9].value[1] := -1;
                threat_decon[18, 10].value[1] := -1;
                threat_decon[19, 1].value[1] := -1;
                threat_decon[19, 2].value[1] := -1;
                threat_decon[19, 3].value[1] := -1;
                threat_decon[19, 7].value[1] := -1;
                threat_decon[19, 8].value[1] := -1;
                threat_decon[19, 9].value[1] := -1;
                threat_decon[19, 10].value[1] := -1;
```

```
                threat_decon[20, 1].value[1] := -1;
                threat_decon[20, 2].value[1] := -1;
                threat_decon[20, 3].value[1] := -1;
                threat_decon[20, 4].value[1] := -1;
                threat_decon[20, 6].value[1] := -1;
                threat_decon[20, 7].value[1] := -1;
                threat_decon[20, 8].value[1] := -1;
                threat_decon[20, 9].value[1] := -1;
                threat_decon[20, 10].value[1] := -1;
                end;
        end
else
        begin
        if (x_max = 10) then
                begin
                threat[9, 1] := -1;
                threat[9, 5] := -1;
                threat[10, 1] := -1;
                threat[10, 2] := -1;
                threat[10, 4] := -1;
                threat[10, 5] := -1;
                end
        else
                begin
                threat[16, 10] := -1;
                threat[17, 1] := -1;
                threat[17, 9] := -1;
                threat[17, 10] := -1;
                threat[18, 1] := -1;
                threat[18, 2] := -1;
                threat[18, 8] := -1;
                threat[18, 9] := -1;
                threat[18, 10] := -1;
                threat[19, 1] := -1;
                threat[19, 2] := -1;
                threat[19, 3] := -1;
```

```
                    threat[19, 7] := -1;
                    threat[19, 8] := -1;
                    threat[19, 9] := -1;
                    threat[19, 10] := -1;
                    threat[20, 1] := -1;
                    threat[20, 2] := -1;
                    threat[20, 3] := -1;
                    threat[20, 4] := -1;
                    threat[20, 6] := -1;
                    threat[20, 7] := -1;
                    threat[20, 8] := -1;
                    threat[20, 9] := -1;
                    threat[20, 10] := -1;
                    end;
            end;
    end;

(* save_best_block_paths stores the two best block paths and scores *)

procedure save_best_block_paths;

    var
            i : integer;
    begin
    for i := 1 to x_max do
            begin
            block[1].x[i] := best_xpath[i];
            block[1].y[i] := best_ypath[i];
            block[2].x[i] := next_best_xpath[i];
            block[2].y[i] := next_best_ypath[i];
            end;
    block[1].score := best_score;
    block[2].score := next_best_score;
    end;

(* new_best replaces the current best/next best block path *)
```

```
(* with a better block path                                    *)

procedure new_best(rank : integer);

    var
            i : integer;
    begin
    if rank = 1 then
            (* new best block path found *)
            begin
            num_ties := 0;
            next_best_score := best_score;
            best_score := cur_score;
            for i := 1 to x_max do
                    (* replace best path with current path and *)
                    (* replace next best with old best path    *)
                    begin
                    next_best_xpath[i] := best_xpath[i];
                    next_best_ypath[i] := best_ypath[i];
                    best_xpath[i] := cur_xpath[i];
                    best_ypath[i] := cur_ypath[i];
                    end;
            end
    else
            (* new next best path found *)
            begin
            next_best_score := cur_score;
            for i := 1 to x_max do
                    (* replace next best path with current path *)
                    begin
                    next_best_xpath[i] := cur_xpath[i];
                    next_best_ypath[i] := cur_ypath[i];
                    end;
            end;
    end;
```

```
(* score_plus adds a threat value to the score of the current   *)
(* block path if it has not already been added from another box *)

function score_plus(score, x, y : integer) : integer;

    var
            i, k : integer;
    begin
    if threat_decon[x, y].num <> 0 then
            (* there are threat values in this box *)
            begin
            for i := 1 to threat_decon[x, y].num do
                    (* process each threat value for the box *)
                    begin
                    k := threat_decon[x,-y].kind[i];
                    if threat_added.add[k] = 0 then
                            (* score has not yet been added *)
                            begin
                            score := score + threat_decon[x, y].value[i];
                            threat_added.add[k] := 1;
                            threat_added.x[k] := x;
                            threat_added.y[k] := y;
                            end;
                    end;
            end;
    score_plus := score;
    end;

(* score_minus subtracts a threat value from the score of a block *)
(* path before the block path continues in another direction      *)

function score_minus(score, x, y : integer) : integer;

    var
            i, k : integer;
    begin
```

```
    if threat_decon[x, y].num <> 0 then
            (* there are threat values in this box *)
            begin
            for i := 1 to threat_decon[x, y].num do
                    (* process each threat value for the box *)
                    begin
                    k := threat_decon[x, y].kind[i];
                    if (threat_added.add[k] = 1) and
                       (threat_added.x[k] = x) and
                       (threat_added.y[k] = y) then
                            (* subtract threat value from score *)
                            (* and mark as not added            *)
                            begin
                            score := score - threat_decon[x, y].value[i];
                            threat_added.add[k] := 0;
                            threat_added.x[k] := 0;
                            threat_added.y[k] := 0;
                            end;
                    end;
            end;
    score_minus := score;
    end;

(* save_tie saves the current block path that has the *)
(* same score as the best block path so far          *)

procedure save_tie;
    var
            i : integer;
    begin
    for i := 1 to x_max do
            begin
            tie_paths[num_ties].x[i] := cur_xpath[i];
            tie_paths[num_ties].y[i] := cur_ypath[i];
            end;
    end;
```

```
                max_pt := best_ypath[i] * box_size;
                min_pt := max_pt - (2 * box_size);
                end;
        if best_ypath[i] < best_ypath[i + 1] then
                begin
                max_pt := best_ypath[i + 1] * box_size;
                min_pt := max_pt - (2 * box_size);
                end;
        if best_ypath[i] = best_ypath[i + 1] then
                begin
                max_pt := best_ypath[i] * box_size;
                min_pt := max_pt - box_size;
                end;
        segment := (max_pt - min_pt) / (pts_on_line - 1);
        for j := 1 to pts_on_line do
                (* compute each point for a box division *)
                begin
                if ((num_div = 10) or (x_max = 20)) then
                        begin
                        next_pt.xcoor[i, j] := i * box_size;
                        next_pt.ycoor[i, j] := min_pt + ((j - 1) * segment);
                        end
                else
                        begin
                        next_pt.xcoor[i*2-1, j] := (i * box_size) - (box_size div 2) ;
                        next_pt.ycoor[i*2-1, j] := ((best_ypath[i]*box_size)-box_size)
                                +((j-1)*(x_max/(pts_on_line-1)));
                        next_pt.xcoor[i*2, j] := i * box_size;
                        next_pt.ycoor[i*2, j] := min_pt+((j-1)*segment);
                        end;
                end;
        end;
if ((num_div = 20) and (x_max = 10)) then
        begin
        for j := 1 to pts_on_line do
                begin
```

```
                                        save_path(i, j, k);
                                        end;
                                end;
                        end; (* k loop *)
                end; (* j loop *)
        restore_path(i);
        end; (* i loop *)
    for i := 1 to pts_on_line do (* last leg *)
        begin
        dist := distance(next_pt.xcoor[num_div - 1, i], next_pt.ycoor[num_div - 1, i],
                        last_xpt, last_ypt) + path.len[i, num_div - 1];
        score := conflict_val(next_pt.xcoor[num_div - 1, i],
                              next_pt.ycoor[num_div - 1, i], last_xpt,
                              last_ypt) + path.conflict[i, num_div - 1];
        path.xpt[i, num_div] := next_pt.xcoor[num_div - 1, i];
        path.ypt[i, num_div] := next_pt.ycoor[num_div - 1, i];
        path.conflict[i, num_div] := score;
        path.len[i, num_div] := dist;
        end;
    end;

(* figure_pts computes all the points on each division *)
(* line for each box in the block path                *)

procedure figure_pts;

    var
        i, j, box_size : integer;
        min_pt, max_pt, segment : real;
    begin
    box_size := x_limit div x_max;
    for i := 1 to x_max - 1 do
        (* find the min and max points for each box division *)
        begin
        if best_ypath[i] > best_ypath[i + 1] then
                begin
```

```
begin
dist := distance(next_pt.xcoor[i, k],
                 next_pt.ycoor[i, k],
                 next_pt.xcoor[i + 1, j],
                 next_pt.ycoor[i + 1, j]) +
        path.len[k, i];
score := conflict_val(next_pt.xcoor[i, k],
                      next_pt.ycoor[i, k],
                      next_pt.xcoor[i + 1, j],
                      next_pt.ycoor[i + 1, j])
                      + path.conflict[k, i];
if score <= temp_score then
        begin
        if score = temp_score then
                (* tie score - check distance *)
                begin
                if dist < temp_dist then
                        begin
                        path.xpt[j, i + 1] := next_pt.xcoor[i, k];
                        path.ypt[j, i + 1] := next_pt.ycoor[i, k];
                        path.conflict[j, i + 1] := score;
                        path.len[j, i + 1] := dist;
                        temp_score := score;
                        temp_dist := dist;
                        save_path(i, j, k);
                        end;
                end
        else
                (* better score - save path *)
                begin
                path.xpt[j, i + 1] := next_pt.xcoor[i, k];
                path.ypt[j, i + 1] := next_pt.ycoor[i, k];
                path.conflict[j, i + 1] := score;
                path.len[j, i + 1] := dist;
                temp_score := score;
                temp_dist := dist;
```

```
                    begin
                    path.xpt[i, j] := temp_path.xpt[i, j];
                    path.ypt[i, j] := temp_path.ypt[i, j];
                    path.conflict[i, j] := temp_path.conflict[i, j];
                    path.len[i, j] := temp_path.len[i, j];
                    end;
            end;
    end;

(* compute_flight_paths stores all flight paths within a *)
(* block path including their costs and lengths          *)

procedure compute_flight_paths;

    var
            i, j, k : integer;
            dist, score, temp_dist, temp_score : real;
    begin
    for i := 1 to pts_on_line do (* first leg *)
            begin
            dist := distance(first_xpt, first_ypt, next_pt.xcoor[1, i],
                             next_pt.ycoor[1, i]);
            score := conflict_val(first_xpt, first_ypt, next_pt.xcoor[1, i],
                                  next_pt.ycoor[1, i]);
            path.xpt[i, 1] := first_xpt;
            path.ypt[i, 1] := first_ypt;
            path.conflict[i, 1] := score;
            path.len[i, 1] := dist;
            end;
    for i := 1 to num_div - 2 do (* middle legs *)
            begin
            for j := 1 to pts_on_line do (* for each of the points on a division *)
                    begin
                    temp_score := 10000.0;
                    temp_dist := 10000.0;
                    for k := 1 to pts_on_line do (* for each of the points on the next division *)
```

```
                    else if (p2_to_center <= sqrt(p1_to_center ** 2 + p1_to_p2 ** 2)) and
                            (p1_to_center <= sqrt(p2_to_center ** 2 + p1_to_p2 ** 2)) and
                            (p1_to_center >= threat_in.radius[i]) then
                            cost := cost + threat_in.value[i];
                    end;
            end;
    conflict_val := cost;
    end;

(* save_path stores the current path so far *)
(* into a temporary array                  *)

procedure save_path(path_len, path_to, path_from : integer);

    var
            i : integer;
    begin
    for i := 1 to path_len do
            begin
            temp_path.xpt[path_to, i] := path.xpt[path_from, i];
            temp_path.ypt[path_to, i] := path.ypt[path_from, i];
            temp_path.conflict[path_to, i] := path.conflict[path_from, i];
            temp_path.len[path_to, i] := path.len[path_from, i];
            end;
    end;

(* restore_path places a stored path back into the current path array *)

procedure restore_path(path_len : integer);

    var
            i, j : integer;
    begin
    for i := 1 to pts_on_line do
            begin
            for j := 1 to path_len do
```

```
                end;
            end;
    end;

(* conflict_val computes the threat cost between two points *)
(*                                                          *)
(* NOTE  except for a few differences, this routine is from *)
(*       program PATH_FINDER written by Carl Lizza and is   *)
(*       used here with his permission                      *)

function conflict_val(x1, y1, x2, y2 : real) : real;

    var
            i : integer;
            cost, p1_to_p2, x1_to_x2, y1_to_y2, z, center_to_line,
              p1_to_center, p2_to_center : real;
    begin
    cost := 0.0;
    p1_to_p2 := distance(x1, y1, x2, y2);
    x1_to_x2 := x1 - x2;
    y1_to_y2 := y1 - y2;
    z := x1_to_x2 * y1 - y1_to_y2 * x1;
    for i := 1 to threat_num do
            (* check points against each threat *)
            begin
            center_to_line := abs(y1_to_y2 * threat_in.cx[i] -
                                  x1_to_x2 * threat_in.cy[i] + z) / p1_to_p2;
            p1_to_center := distance(x1, y1, threat_in.cx[i], threat_in.cy[i]);
            p2_to_center := distance(x2, y2, threat_in.cx[i], threat_in.cy[i]);
            if center_to_line < threat_in.radius[i] then
                    (* does threat cover an endpoint *)
                    begin
                    if ((p2_to_center < threat_in.radius[i]) and
                        (p1_to_center >= threat_in.radius[i])) or
                       ((x1 = 0.0) and ( p1_to_center <= threat_in.radius[i])) then
                            cost := cost + threat_in.value[i]
```

```
    flight[path_num].cost := path.conflict[inx, num_div];
    flight[path_num].len := path.len[inx, num_div];
    end;

(* compute_best_flight_path locates the best flight path generated *)

procedure compute_best_flight_path(var inx : integer);

    var
            score, dist : real;
            i : integer;
    begin
    score := path.conflict[1, num_div];
    dist := path.len[1, num_div];
    inx := 1;
    for i := 2 to pts_on_line do
            (* process each path *)
            begin
            if path.conflict[i, num_div] <= score then
                    (* best path so far *)
                    begin
                    if path.conflict[i, num_div] = score then
                            begin
                            if path.len[i, num_div] < dist then
                                    begin
                                    inx := i;
                                    dist := path.len[i, num_div];
                                    score := path.conflict[i, num_div];
                                    end;
                            end
                    else
                            begin
                            inx := i;
                            dist := path.len[i, num_div];
                            score := path.conflict[i, num_div];
                            end;
```

```
            next_best_xpath[i] := tie_paths[inx2].x[i];
            next_best_ypath[i] := tie_paths[inx2].y[i];
            end;
        next_best_score := best_score;
        end;


        begin  (* main section of block_paths *)
        init_block_paths;
        best_block_paths(x, y);
        if num_ties > 0 then
                resolve_ties;
        save_best_block_paths;
        end;

(* exhaust performs an exhaustive search of block paths *)
(* to find the best flight path                         *)

procedure exhaust;

        var
                inx : integer;
                temp_path : flight_paths;

(* save_flight saves a flight path, cost, and length *)

procedure save_flight(path_num, inx : integer);

        var
                i : integer;
        begin
        for i := 1 to num_div do
                begin
                flight[path_num].x[i] := path.xpt[inx, i];
                flight[path_num].y[i] := path.ypt[inx, i];
                end;
```

```
(* resolve_ties considers all block paths with the same best *)
(* scores and chooses the two that are the most different    *)

procedure resolve_ties;
    var
            i, j, diff, best_diff, inx1, inx2 : integer;
    begin
    (* put best block path with the ties *)
    num_ties := num_ties + 1;
    inx1 := 1;
    inx2 := 2;
    for i := 1 to x_max do
            begin
            tie_paths[num_ties].x[i] := best_xpath[i];
            tie_paths[num_ties].y[i] := best_ypath[i];
            end;
    best_diff := 0;
    (* find the two paths most different *)
    for i := 1 to num_ties - 1 do
            begin
            for j := i + 1 to num_ties do
                    begin
                    diff := compute_diff(i, j);
                    if diff > best_diff then
                            begin
                            best_diff := diff;
                            inx1 := i;
                            inx2 := j;
                            end;
                    end;
            end;
    (* store the two most different *)
    for i := 1 to x_max do
            begin
            best_xpath[i] := tie_paths[inx1].x[i];
            best_ypath[i] := tie_paths[inx1].y[i];
```

```
                        best_block_paths(x, y);
                end;
        end;
    end;

(* best_block_paths is the main driver that checks all block paths *)
(* in the grid to determine the best block paths                   *)

procedure best_block_paths;

    begin
    path_up(x, y);
    path_next(x, y);
    path_down(x, y);
    visit[x, y] := 0;
    if option = DECONFLICT then
            cur_score := score_minus(cur_score, x, y)
    else
            cur_score := cur_score - threat[x, y];
    inx := inx - 1;
    end;

(* compute_diff computes the difference between the y coordinates *)
(* of two block paths and returns the score                       *)

function compute_diff(inx1, inx2 : integer) : integer;
    var
            score, i : integer;
    begin
    score := 0;
    for i := 1 to x_max do
            score := score + abs(tie_paths[inx1].y[i] -
                                 tie_paths[inx2].y[i]);
    compute_diff := score;
    end;
```

```
 (threat_decon[x + 1, y - 1].value[1] <> -1) and
 (visit[x + 1, y - 1] = 0)) or
((option = ALL) and (threat[x + 1, y - 1] >= 0) and
 (visit[x + 1, y - 1] = 0)) then
    (* box available for processing and has *)
    (* not been visited on current path     *)
    begin
    x := x + 1;
    y := y - 1;
    if (option = DECONFLICT) then
            cur_score := score_plus(cur_score, x, y)
    else
            cur_score := cur_score + threat[x, y];
    add_to_list(x, y);
    visit[x, y] := 1;
    if ((x_max = 10) and (x = 10) and (y = 3)) or
       ((x_max = 20) and (x = 20) and (y = 5)) then
            (* have reached goal for current path *)
            begin
            if (cur_score = best_score) and (num_ties < max_ties - 1) then
                    begin
                    num_ties := num_ties + 1;
                    save_tie;
                    end;
            if cur_score < best_score then
                    new_best(1)
            else if cur_score < next_best_score then
                    new_best(2);
            visit[x, y] := 0;
            if option = DECONFLICT then
                    cur_score := score_minus(cur_score, x, y)
            else
                    cur_score := cur_score - threat[x, y];
            inx := inx - 1;
            end
     else (* continue processing current block path *)
```

```
                    add_to_list(x, y);
                    visit[x, y] := 1;
                    if ((x_max = 10) and (x = 10) and (y = 3)) or
                       ((x_max = 20) and (x = 20) and (y = 5)) then
                            (* have reached goal for current path *)
                            begin
                            if (cur_score = best_score) and (num_ties < max_ties - 1) then
                                    begin
                                    num_ties := num_ties + 1;
                                    save_tie;
                                    end;
                            if cur_score < best_score then
                                    new_best(1)
                            else if cur_score < next_best_score then
                                    new_best(2);
                            visit[x, y] := 0;
                            if option = DECONFLICT then
                                    cur_score := score_minus(cur_score, x, y)
                            else
                                    cur_score := cur_score - threat[x, y];
                            inx := inx - 1;
                            end
                    else (* continue processing current block path *)
                            best_block_paths(x, y);
                    end;
            end;
    end;

(* path_down processes the next box down in the current block path *)

procedure path_down(x, y : integer);

    begin
    if ((x + 1 <= x_max) and (y - 1 >= 1)) then
            begin
            if ((option = DECONFLICT) and
```

```
                                new_best(1)
                            else if cur_score < next_best_score then
                                new_best(2);
                            visit[x, y] := 0;
                            if option = DECONFLICT then
                                cur_score := score_minus(cur_score, x, y)
                            else
                                cur_score := cur_score - threat[x, y];
                            inx := inx - 1;
                            end
                        else (* continue processing current block path *)
                            best_block_paths(x, y);
                        end;
                end;
        end;

(* path_next processes the box next to the current box *)

procedure path_next(x, y : integer);

    begin
    if (x + 1 <= x_max) then
            begin
            if ((option = DECONFLICT) and
                (threat_decon[x + 1, y].value[1] <> -1) and
                (visit[x + 1, y] = 0)) or
               ((option = ALL) and (threat[x + 1, y] >= 0) and
                (visit[x + 1, y] = 0)) then
                    (* box available for processing and has *)
                    (* not been visited on current path     *)
                    begin
                    x := x + 1;
                    if (option = DECONFLICT) then
                            cur_score := score_plus(cur_score, x, y)
                    else
                            cur_score := cur_score + threat[x, y];
```

```
procedure best_block_paths(x, y : integer); forward;

(* path_up processes the next box up in the current block path *)

procedure path_up(x, y : integer);

    begin
    if ((x + 1 <= x_max) and (y + 1 <= y_max)) then
            begin
            if ((option = DECONFLICT) and
                (threat_decon[x + 1, y + 1].value[1] <> -1) and
                (visit[x + 1, y + 1] = 0)) or
               ((option = ALL) and (threat[x + 1, y + 1] >= 0) and
                (visit[x + 1, y + 1] = 0)) then
                    (* box available for processing and has *)
                    (* not been visited on current path     *)
                    begin
                    x := x + 1;
                    y := y + 1;
                    if (option = DECONFLICT) then
                            cur_score := score_plus(cur_score, x, y)
                    else
                            cur_score := cur_score + threat[x, y];
                    add_to_list(x, y);
                    visit[x, y] := 1;
                    if ((x_max = 10) and (x = 10) and (y = 3)) or
                       ((x_max = 20) and (x = 20) and (y = 5)) then
                            (* have reached goal for current path *)
                            begin
                            if (cur_score = best_score) and (num_ties < max_ties - 1) then
                                    begin
                                    num_ties := num_ties + 1;
                                    save_tie;
                                    end;
                            if cur_score < best_score then
```

```
                    next_pt.xcoor[num_div-1, j] := (x_max*box_size)-(box_size div 2);
                    next_pt.ycoor[num_div-1, j] := ((best_ypath[x_max]*box_size)-box_size)
                            + ((j-1)*(x_max/(pts_on_line-1)));
                    end;
            end;
    end;


    begin  (* main section of exhaust *)
    path_num := path_num + 1;
    figure_pts;
    compute_flight_paths;
    compute_best_flight_path(inx);
    save_flight(path_num, inx);
    end;

(* next_flight_path transfers the next path into best_path *)
(* for processing by exhaust                               *)

procedure next_flight_path;

    var
            i : integer;
    begin
    for i := 1 to x_max do
            begin
            best_xpath[i] := next_best_xpath[i];
            best_ypath[i] := next_best_ypath[i];
            end;
    end;

(* output_paths chooses the best flight path and its *)
(* block path and prints the final results           *)

procedure output_paths;
```

```
var
        i, inx : integer;
begin
inx := 1;
if (flight[inx + 1].cost < flight[inx].cost) or
      ((flight[inx + 1].cost = flight[inx].cost) and
   (flight[inx + 1].len < flight[inx].len)) then
        inx := 2;
tot_cpu_time := clock - tot_cpu_time;  (* system clock *)
(* print block path and score *)
writeln(pathout);
writeln(pathout, 'BLOCK PATH');
writeln(pathout);
writeln(pathout, '          X          Y');
writeln(pathout);
for i := 1 to x_max do
        writeln(pathout, block[inx].x[i], block[inx].y[i]);
writeln(pathout);
writeln(pathout, 'SCORE = ', block[inx].score : 4);
writeln(pathout);
(* print flight path, cost, and length *)
writeln(pathout);
writeln(pathout, 'FLIGHT PATH');
writeln(pathout);
writeln(pathout, '          X          Y');
writeln(pathout);
for i := 1 to num_div do
        writeln(pathout, flight[inx].x[i] :10:2, flight[inx].y[i] :10:2);
writeln(pathout, last_xpt :10:2, last_ypt :10:2);
writeln(pathout);
writeln(pathout, 'COST = ', flight[inx].cost : 6:2);
writeln(pathout);
writeln(pathout, 'LENGTH = ', flight[inx].len : 6:2);
writeln(pathout);
writeln(pathout, 'CPU TIME(msec) = ', tot_cpu_time);
end;
```

```
begin  (* main section of hplan *)
initialize;
read_data;
process_threats;
block_paths;
exhaust;
next_flight_path;
exhaust;
output_paths;
end.
```

APPENDIX B

```
(* program threat_bldr builds output file datain for use by program *)
(* hplan and contains run options and a random threat environment   *)
(*                                                                  *)
(*    NOTE  except for a few differences, this program was written  *)
(*          by Carl Lizza and is used here with his permission      *)

program threat_bldr(input, output, datain);

    const
            pi = 3.1415927;
            x_limit = 100;
            y_limit = 50;

    type
            block_size = (BIG, SMALL);
            cost_option = (ALL, DECONFLICT);
            coverage = (CENTER, TOTAL);

    var
            i, j, nr_threats, count, x_max, y_max,
              seed, num_div, pts_on_line, cost : integer;
            grid_area, total_density, radius,
              threat_density, x_coord, y_coord : real;
            type_path : block_size;
            option : cost_option;
            box_coverage : coverage;
            datain : text;

(* rand computes a random number *)

function rand(var seed : integer) : real;
```

```
begin
seed := ((25173 * seed) + 13849) mod 65536;
rand := seed / 65536;
end;


begin  (* main section of threat_bldr *)
rewrite(datain);
writeln('ENTER TYPE OF BLOCK PATH -- BIG or SMALL');
readln(type_path);
writeln(datain, type_path);
writeln('ENTER COST OPTION -- ALL or DECONFLICT');
readln(option);
writeln(datain, option);
writeln('ENTER BOX THREAT COVERAGE -- CENTER or TOTAL');
readln(box_coverage);
writeln(datain, box_coverage);
writeln('ENTER MAXIMUM XY BLOCK COORDINATES -- XMAX YMAX');
readln(x_max, y_max);
writeln(datain, x_max, y_max);
writeln('ENTER NUMBER OF DIVISIONS AND THE NUMBER OF');
writeln('POINTS FOR EACH DIVISION -- NUMDIV PTS');
readln(num_div, pts_on_line);
writeln(datain, num_div, pts_on_line);
writeln('ENTER SEED');
readln(seed);
grid_area := x_limit * y_limit;
writeln('ENTER NUMBER OF THREAT CATEGORIES');
readln(nr_threats);
writeln('ENTER TOTAL THREAT DENSITY');
readln(total_density);
for i := 1 to nr_threats do
        begin
        writeln('FOR THREAT CATEGORY', i : 3, ' ENTER RADIUS, COST, DENSITY');
        readln(radius, cost, threat_density);
        count := trunc(total_density * threat_density * grid_area /
```

```
                    (pi * radius ** 2));
        for j := 1 to count do
                begin
                x_coord := x_limit * rand(seed);
                y_coord := y_limit * rand(seed);
                writeln(datain, x_coord, y_coord, cost, radius);
                end;
        end;
writeln(datain, -1, -1, -1, -1);
end.
```

BIBLIOGRAPHY

1. Fikes, R.E. & N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, pp. 189-208, 1971.

2. Fikes, R.E., P.E. Hart, & N.J. Nilsson, "Learning and Executing Generalized Robot Plans," Artificial Intelligence, Vol. 3, No. 4, pp. 251-288, 1972.

3. Georgeff, M.P., "Strategies in Heuristic Search," Artificial Intelligence, Vol. 20, pp. 393-425, 1983.

4. Grove, D., "Documentation for the FPG Model," University of Dayton, 1980.

5. Lizza, C.S., "Generation of Flight Paths Using Heuristic Search," Wright State University, 1984.

6. McLaughlin, R.G., "Description and Use of SNOOPER III, A Model for Determining the Strategy Needed Over Optimum PEnetration Routes," Cornell Aeronautical Laboratory, 1971.

7. Nilsson, N.J., Principles of Artificial Intelligence, Tioga, Palo Alto, Calif., 1980.

8. Ralston, A., Encyclopedia of Computer Science, Van Nostrand Reinhold, New York, 1976.

BIBLIOGRAPHY (CONTINUED)

9. Rich, E., Artificial Intelligence, McGraw-Hill, New York, 1983.

10. Sacerdoti, E.D., "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, Vol. 5, pp. 115-135, 1974.

11. Sacerdoti, E.D., A Structure for Plans and Behavior, Elsevier, New York, 1977.

12. Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," Artificial Intelligence, Vol. 16, No. 2, pp. 111-139, 1981.

13. Stefik, M., "Planning and Meta-Planning (MOLGEN: Part 2)," Artificial Intelligence, Vol. 16, No. 2, pp. 141-169, 1981.

14. Wilkins, D., "Using Patterns and Plans in Chess," Artificial Intelligence, Vol. 14, pp. 165-203, 1980.

GENERATION OF FLIGHT PATHS

USING HIERARCHICAL PLANNING

A thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science

By

KEVIN BRIAN KLINE
B.S., Pennsylvania State University, 1979
A.S., Thomas Nelson Community College, 1977

1985
Wright State University

WRIGHT STATE UNIVERSITY

SCHOOL OF GRADUATE STUDIES

March, 1985

I HEREBY RECOMMEND THAT THE THESIS PREPARED UNDER MY SUPERVISION BY Kevin Brian Kline ENTITLED Generation of Flight Paths Using Hierarchical Planning BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF SCIENCE .

Thomas A. Sudkamp
Thesis Director

Chairman of Department

Committee on
Final Examination

Thomas A Sudkamp

Dean of the School of Graduate Studies

ABSTRACT

Kline, Kevin Brian. M.S., Department of Computer Science, Wright State University, 1985. Generation of Flight Paths Using Hierarchical Planning.

This thesis examines the use of an artificial intelligence technique, hierarchical planning, to solve the problem of generating an aircraft route and finding a path through various hostile environments. A route or path, is evaluated by the number and type of threats the aircraft encounters on the route and the route length. An algorithm using hierarchical planning is presented and tested against several hostile environments. Specifically, the algorithm will divide the problem space or grid, into smaller spaces or boxes. These boxes are then assigned values based upon the input hostile environment. Block paths are then constructed and evaluated based on the values in the boxes. An exhaustive search is performed on the two best block paths to find a flight path for the aircraft. Test results are compared to previous results obtained using heuristic search and indicate an improvement in solution quality. Although specific plans are incorporated into the algorithm to obtain test results, many other plans within the realm of hierarchical planning certainly exist and could be used to solve this problem.

TABLE OF CONTENTS

TABLE OF CONTENTS (CONTINUED)

LIST OF FIGURES

details of the original problem are introduced and this process continues until the original problem is solved. Sacerdoti [10] terms this process as "length-first" search. This is because the method uses an abstraction space and follows through to the original goal state before exploring the next level of abstraction or computing the final solution. This enables the program to find any steps that will possibly lead to dead ends or undesirable solutions.

The technique of hierarchical planning can be illustrated by an example of finding a suitable travel route from Dayton, Ohio to San Antonio, Texas. Instead of being forced to consider the details such as route numbers, consider the subproblem of going from the state of Ohio to Texas. In order to ignore further details at this time, plot the route from Ohio to Texas by going state to state. Now the subproblem to be solved is to find a route from Dayton, Ohio to San Antonio, Texas through Indiana, Illinois, Missouri, and Oklahoma. This process may continue to solve additional subproblems or go directly to the final solution. The main point of this example is the total search space, all routes in the United States, has been reduced to just the routes that are contained in the solution for the subproblem. Additionally, the details of looking at individual routes was not introduced immediately, but ignored until they are needed.

The problem-solving system NOAH [11] also performs planning in a hierarchical fashion. It does this by first constructing an abstract skeleton of a plan and then successively considers more details. NOAH is presented a problem as a statement that is to be made true by applying a sequence of actions to a given initial state. Then it introduces another level of detail of the solution and solves for more

IV. THE HIERARCHICAL PLANNING MODEL

Hierarchical planning is a techinique that divides a problem into smaller subproblems, with the important capability to defer consideration of the details of a problem at the higher level plan. The crucial point is to discriminate between the important information and details of the problem space. Details of the problem are essentially ignored at the higher level plan or plans and are only introduced when the subproblems are solved. This approach delays consideration of some details until the problem search space has been significantly reduced by solving the subproblems at a higher level. The details of the problem are accounted for after a solution is found in an abstract problem space, thereby increasing program power and efficiency.

THE HIERARCHICAL PLANNING TECHNIQUE

For complicated problems, one technique used is to work on small pieces of the problem and combine their solutions into a complete solution for the original problem. The planning approach involves decomposing or breaking down the original problem into appropriate subparts. Decomposition involves dividing the original problem into several subproblems. In the hierarchical planning technique, the original problem space is redefined into one or more abstraction spaces. Sacerdoti [10] describes an abstraction space as a simplifying representation of the problem space in which unimportant details are ignored. These abstraction spaces can then be searched to solve simple, more manageable subproblems. As these subproblems are solved, more

contain weights that may be adjusted. The problem is in obtaining the right combination of weights. A problem encountered with this method is computer run time relating to leg length. As leg length is decreased, the total number of legs generated is increased. Because actual costs are computed and heuristic evaluation for an estimate of future costs is required for each leg, computer time is increased significantly. For example, based on the results in [5], when leg length was reduced by one half, CPU time increased by three to one hundred times.

Comparing the two models above, SNOOPER looks at the total threat environment to determine the best path but has a computational explosion because it examines every possible move. FPG is only concerned with evaluating a single leg at a time and therefore does not take advantage of future threat analysis. One possible improvement is to use a technique to cut down on the search space while taking into account the entire threat environment. This leads to the next method discussed, heuristic search.

HEURISTIC SEARCH MODEL

The heuristic search model [5] defines the problem space in much the same way as the two previous methods. There is a grid defined with an x and y axis and threats located throughout the grid. The model attempts to find a solution to the problem by using a modified A* search algorithm. An A* algorithm is a method that uses heuristics, knowledge applied to an algorithm to minimize search space, to direct the path search in an 'intelligent' manner. The goal is to limit the number of paths the algorithm must search through and at the same time find the best path. Paths consist of legs which are generated based on user input. These inputs define the path deflection angle, which is equivalent to the threat detection wedge in the FPG model, the length of a leg, and the number of legs in the path detection angle. The important feature of this method are the heuristics. Heuristics are used to estimate the future costs to the goal. The methods of cost computation are detailed in [5]. This method uses heuristics to cut down on the total search space and determine a path based on the heuristics. The actual makeup of the heuristics is paramount with this method. Poor heuristics may produce bad results. The heuristics used

and number of legs from a point are also aircraft-dependent. Together these parameters define the look-ahead capability called the threat detection wedge. Figure 3.2 illustrates an aircraft's threat detection

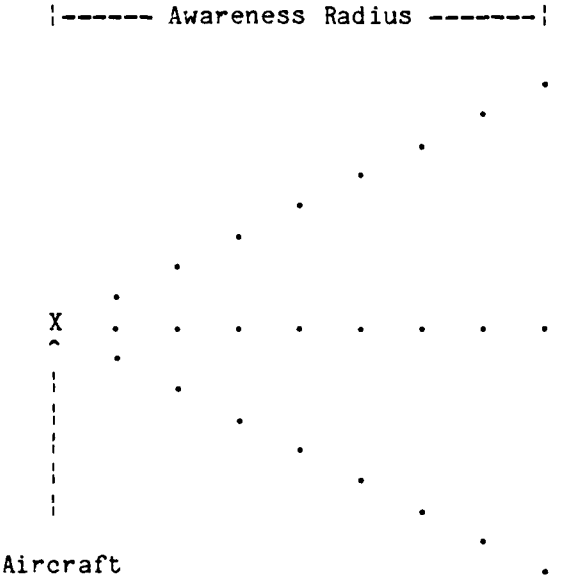


Figure 3.2

Threat Detection Wedge

wedge. The algorithm looks at the possible legs from the current point and determines which leg is best. This determination is made by finding the leg that will encounter the least amount of threat. Threats outside of the awareness wedge are not considered. In other words, future threats on the grid are not considered when determining a current leg. This process continues until the goal state is reached. This model is useful when there is no prior knowledge of the entire threat environment, but undesirable compared to other methods when all the threats are known because it cannot anticipate future threats in the grid.

illustrated in Figure 3.1. Legs between adjacent points incur a cost or

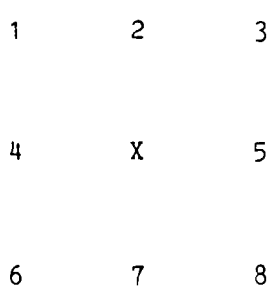


Figure 3.1

8 Possible Legs to X

penalty. The cost is calculated based on the distance between points, the altitude and speed of the aircraft, and threat interaction. The results are saved and used to determine the best path from the goal to the desired starting point. The best path or route is defined as the one that encounters the least amount of threat. Length is also used to determine the best path, but is secondary to the threat cost of the path. This method will find the best path because it examines all possible moves from each point defined in the grid, but this exhaustive search of all the possible points is restrictive because of the computer resources it requires.

FLIGHT PATH GENERATION (FPG) MODEL

The FPG model [4] also uses a grid defined by an x, y axis. A start state and goal state are defined and the algorithm begins at the start state. The distance from the start state to the goal state is a path consisting of several legs. The length of each leg is equal to the distance a particular aircraft can look ahead to see threats. This distance is called the awareness radius of the aircraft. The direction

III. SOLUTION METHODS

This section describes three solution methods for finding a suitable aircraft route through hostile environments. The first two methods described are the Strategy Needed Over Optimum Penetration Routes (SNOOPER) model [6], and the Flight Path Generation (FPG) model [4]. Although only briefly discussed, these two models have influenced the third method which in turn provided an invaluable reference to this thesis. The third method referred to above, Generation of Flight Paths Using Heurisic Search [5], was a thesis recently submitted at Wright State University and will be discussed in depth because it uses another artificial intelligence technique, heuristic search. Another solution method, hierarchical planning, is used in this thesis and will be described in a later chapter.

STRATEGY NEEDED OVER OPTIMUM PENETRATION ROUTES (SNOOPER) MODEL

The SNOOPER model [6] defines the problem space as rectangular in shape and locations in this grid are referenced by x, y coordinates. Within this grid, a point is specified as the goal state. This method searches backward, from the goal state to an undefined starting state. There also exists threat sites within the grid which affect the flight path of the aircraft. The model begins at a fixed goal point on a grid and examines all possible moves to that point and continues this process for each point generated. Penetration route movement from points in the grid proceed to any of eight surrounding points. Movement is either parallel to the x or y axis or at a 45 degree angle. This is

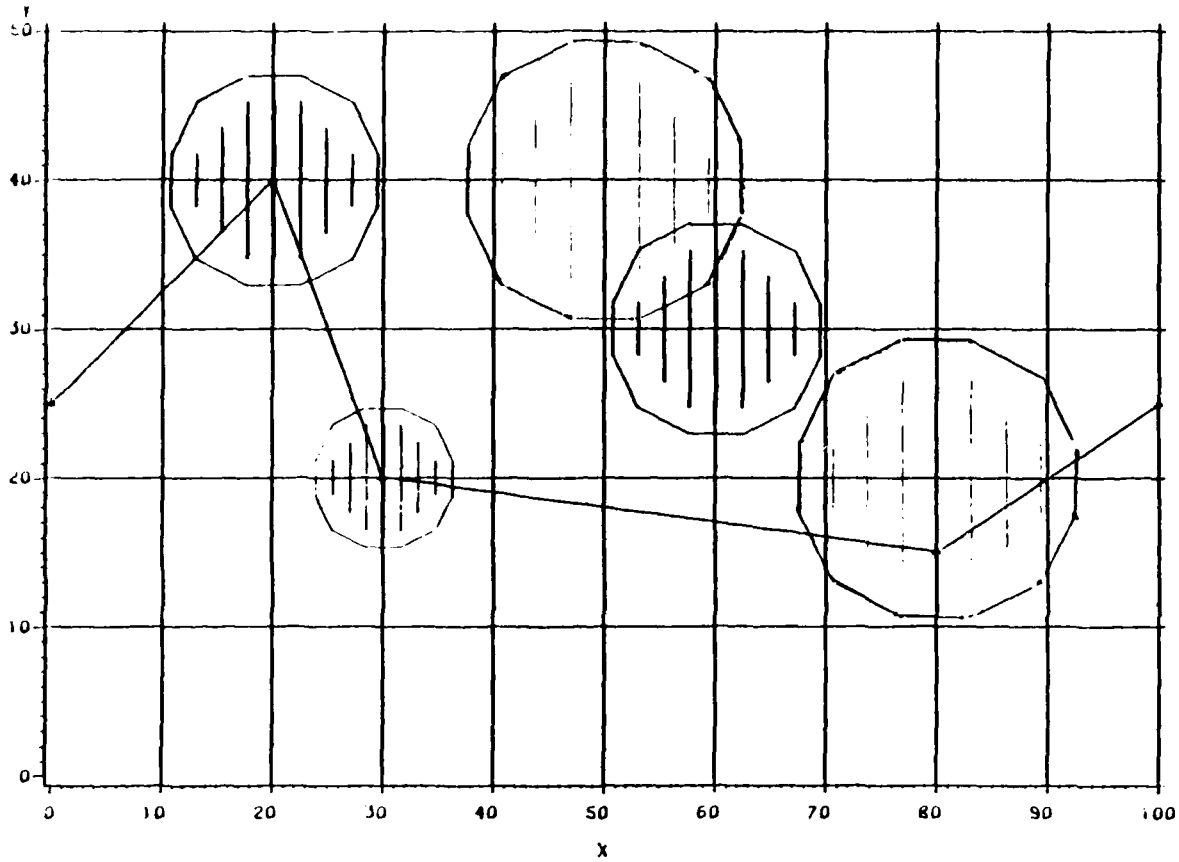


Figure 2.1

Sample Grid, Threats, and Path

threat numbers 1, 3, and 4. The total of the threat costs is 85, which is the cost of the given path.

Threat #	X, Y	Radius	Cost
1	30, 20	5	50
2	50, 40	10	10
3	80, 20	10	10
4	20, 40	7.5	25
5	60, 30	7.5	25

Table 2.1

Sample Threat Values

II. PROBLEM ENVIRONMENT

The problem is to generate an aircraft route through a hostile environment. The problem space is represented by a grid with all points of the grid designated by x and y coordinates. The origin of the grid is in the lower left corner and is designated by the x, y coordinates (0, 0). The aircraft route is defined as starting at the leftmost side of the grid, moving left to right, and ending at the rightmost boundary. A path from a start position to the goal consists of several flight legs. A flight leg is the part of a path connecting two points. The entire grid represents a possible hostile environment. A threat is represented by a circle either partially or entirely within the grid. Each threat location is given by x, y coordinates which indicate its center. There are different threat types, each designated by two components: radius and cost of the threat. The cost of a threat is the cost assigned to a path which intersects the threat. The total cost for a path is the sum of all the costs for the threats the path encounters. Actual computation of the threat cost will be covered in the Hierarchical Planner (HPLAN) implementation. As an example, define a grid with maximum x, y coordinates as (100, 50). The aircraft route or path, is shown as a line starting at (0, 25) and ending at (100, 25). The threats and their costs are given below in Table 2.1. The grid, threats, and path are illustrated in Figure 2.1. The cost of the path shown is 85. This is calculated by adding the values of the threats the path encounters along the way. In this case, the path passes through

I. INTRODUCTION

The objective of this thesis is to study the effectiveness of hierarchical planning to construct a route or path for an aircraft through a hostile environment. An evaluation of a path is based on the number and type of threats an aircraft will encounter on the path. When more than one path has the same cost, the shortest path is considered to be the better path. The use of hierarchical planning for this problem was suggested by Dr. Thomas Sudkamp, Department of Computer Science, Wright State University. A thesis recently submitted under Dr. Sudkamp's supervision [5], approached the same problem using heuristic search. Results from the hierarchical planning method will be compared to the same test cases used in the heuristic search approach.

This paper will define the problem environment and other methods that are used to solve it. The hierarchical planning model will be described along with the planning strategy and how it was developed. The algorithm along with its implementation is presented and test cases with their results will be discussed. Concluding remarks concerning the hierarchical planning method for solving this problem will then be presented.

To Peg, Scott, Chris, and Chalene

The most important people in my life for

their love and support through the years.

LIST OF TABLES

LIST OF FIGURES (CONTINUED)

detailed actions. An example described in [9] and [11] is given to show how NOAH would solve a problem. The problem domain is the blocks world. The blocks world consists of a number of square blocks, all the same size, that can be stacked on each other, and a flat surface on which blocks can be placed. There is a robot arm that can move the blocks. The actions that it can perform and predicates required to specify certain conditions, that are needed for this example are:

ON(A, B):	Block A is on block B.
CLEAR(A):	There is nothing on top of block A.
STACK(A, B):	Will put block A on block B, provided that both objects are clear.

The blocks problem for this example is shown in Figure 4.1. In the initial state, block C is on top of block A and block B is on the table by itself. The final goal is to have block A on block B and also have block B on block C.

NOAH uses a structure called a procedural net, which is a graph structure whose nodes represent actions at varying levels of detail, organized into a hierarchy of partially ordered time sequences. Nodes labeled S indicate a split in the plan and nodes labeled J indicate a join. Square boxes represent operators that will be incorporated into the plan. Boxes with rounded ends denote goals that remain to be satisfied. NOAH also uses a set of "critics" to examine plans and interactions between the subplans. Two examples include critics that are used to resolve conflicts in plans and eliminate redundant specifications of subgoals. Use of these critics will become clear in the discussion of the example. Figure 4.2 illustrates how NOAH solves

the problem presented in Figure 4.1.

The first thing NOAH does is to divide the problem (graph A) into two subproblems (graph B). The preconditions of STACK are now considered and shown in graph C. The nodes in graph C are numbered to aid in explaining actions of the critics. Now the critic, Resolve Conflicts, is invoked and a table is constructed showing literals that must be true for one operation but are also denied by some operation. The table entries are presented below:

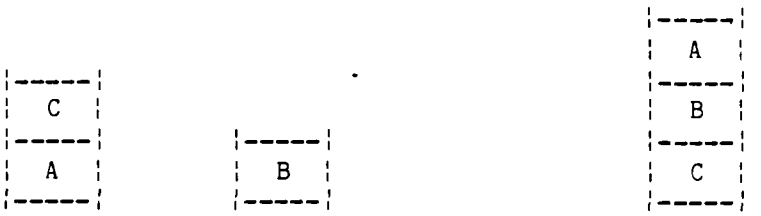
CLEAR(B):	asserted:	node 2 "CLEAR B"
	denied:	node 3 "STACK A on B"
	asserted:	node 4 "CLEAR B"
CLEAR(C):	asserted:	node 5 "CLEAR C"
	denied:	node 6 "STACK B on C"

Sometimes something must be true before an operation is performed, but then will be denied by that same operation. For example, CLEAR(C) denies the STACK(B, C) operation, but the CLEAR(C) protects STACK(B, C). Therefore, those preconditions that are denied by the operation they are protecting are deleted from the table and results in the following table:

CLEAR(B):	denied:	node 3 "STACK A on B"
	asserted:	node 4 "CLEAR B"

This entry contains the fact that since STACK(A, B) will undo the precondition for STACK(B, C), STACK(B, C) should be done first. Graph D shows the plan after this criticism. Now another critic, Eliminate Redundant Preconditions, is invoked. The goal CLEAR(B) appears twice in

graph D. The top CLEAR(B) is redundant because the precondition must exist only for the eariler action, STACK(B, C). This results in graph E. Graph F results from observing that in order to CLEAR(A), C must be removed from A and C must be clear to do that. The Resolve Conflicts critic is used again to ensure that everything depending on C being clear was done prior to STACK(B, C). The result is graph G. The Eliminate Redundant Preconditions critic is used to delete one CLEAR(C) and results in graph H. Next the system observes that CLEAR(C) and CLEAR(B) are true in the initial state. Therefore, the final plan produced by NOAH is in graph I.



Initial State:

ON(C, A) and

CLEAR(B).

Goal:

ON(A, B) and

ON(B, C).

Figure 4.1

Sample Blocks Problem

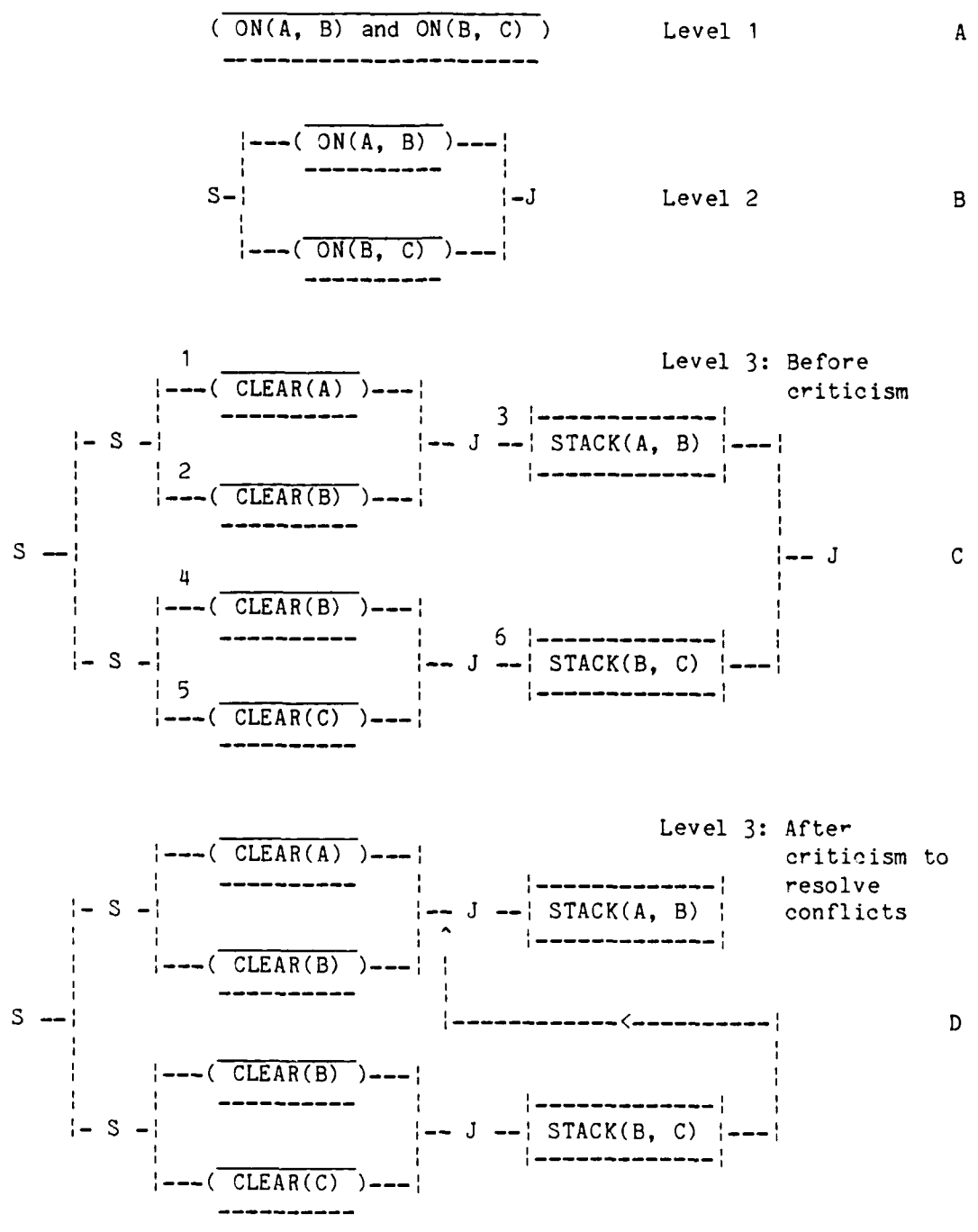


Figure 4.2

NOAH Plans for Sample Problem

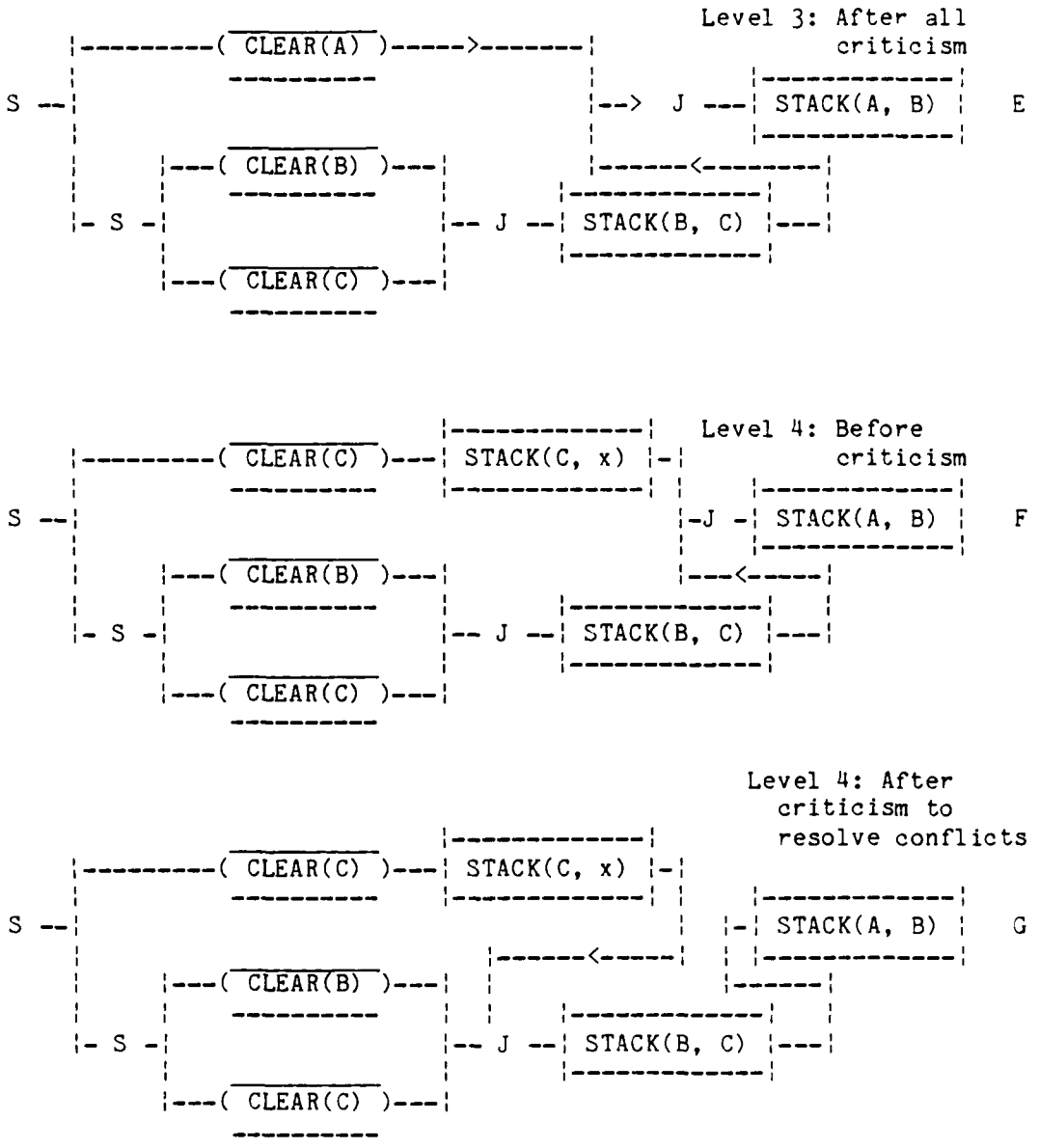


Figure 4.2 -- continued

NOAH Plans for Example Problem

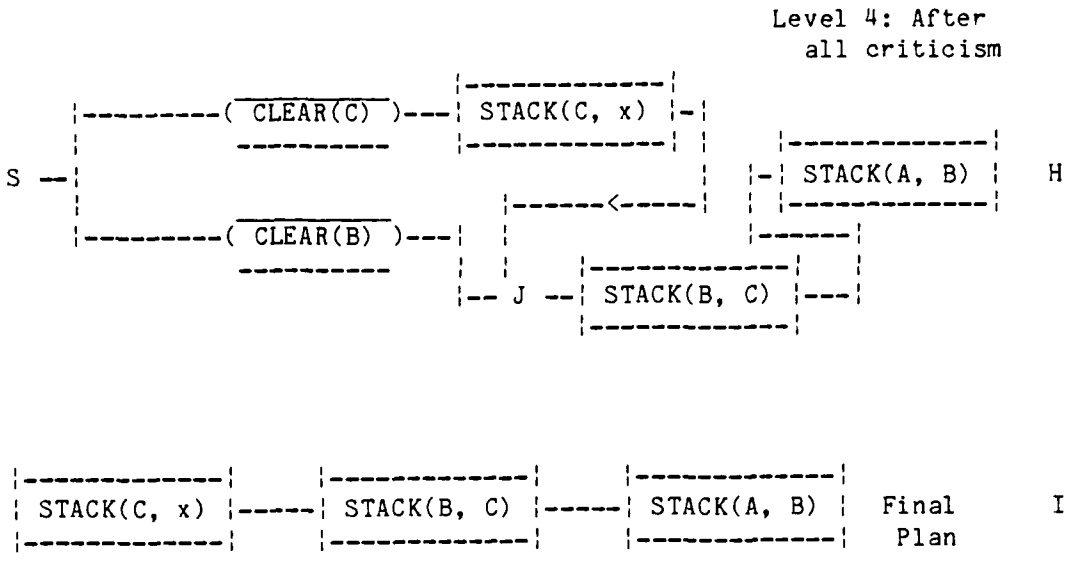


Figure 4.2 -- concluded

NOAH Plans for Sample Problem

MOTIVATION FOR HIERARCHICAL PLANNING

Some computer programs are written to generate all possible solutions and test these solutions to discover the best solution. Many problems are nontrivial and the number of possible solutions and search space are extremely large. Because of computer restraints, such as CPU time and memory, no exhaustive search of the entire problem space is possible. This indicates a need to direct the search through the problem space in order to reduce computer time and memory requirements. Although an improvement over an exhaustive search of the entire problem space, even heuristic search can result in a combinatorial explosion. As noted in [3], even with using heuristic functions, it becomes apparent that planning must be incorporated if substantial reductions in search effort are to be achieved. Nontrivial problems contain many details that need to be taken into consideration in solving the problem and finding the best solution. Attention to these details is exactly the reason for the drawbacks of many problem solvers. The solution is to try to eliminate part of the search space and reduce the original problem to a smaller one and put off consideration of as many details as possible, for as long as possible. This approach would search through an abstraction space, a simplifying representation of the problem space, in which certain details are ignored. When a solution is found to a subproblem in the abstraction space, then details are considered and a final solution to the original problem is found.

The problem solving technique examined in this thesis will be hierarchical planning. This method is used for several reasons. One reason was to observe the similarities and differences between hierarchical planning and heuristic search methods in the same problem

environment. Results from test cases using hierarchical planning will be presented and compared to the results of the heuristic search method obtained in [5]. Another reason this approach to problem solving is used is because it has displayed significant increases in problem-solving power in other systems. The ABSTRIPS problem-solver [10] progressively narrows the search space by solving the problem using simpler, less constrained operators. By solving these simpler problems, Sacerdoti [11] notes that ABSTRIPS is relatively insensitive to combinatorial explosion. In the NOAH system [11], consideration of details in a stepwise fashion increases efficiency because many details of the problem are solved only at lower-level plans. The method of generalizing plans in [2], replacing problem-specific constants with problem-independent parameters, increases its problem-solving capabilities.

V. DETERMINATION OF PLANS

Given the the hierarchical planning method and the problem environment, the next decisions to make are what planning strategy should be used to divide the problem space into subproblems and which details could be ignored at the higher level plans. The first decision made was to divide the problem space into boxes and use an evaluation method to assign values to each box. After this, use a technique to reduce the overall grid size, repeating the process until the search space is small enough to use exhaustive search to find a set of potential solution paths. The details of the problem, such as actual path cost and final path legs, would be considered only after the original search space was small enough to perform the exhaustive search. A decision was made not to use heuristics in this process for two reasons. First, thesis research for this problem using heuristic search was ongoing. Secondly, it was decided that the design should attempt to reduce the search space through hierarchical planning, so that exhaustive search techniques could be applied.

In order to compare final results with those used in the heuristic search method and to keep the problem space consistent throughout the testing phase, the grid is defined to be 100 units wide and 50 units high with starting point at (0, 25) and ending point at (100, 25). Given this grid size, a determination would now be made on how to subdivide the problem space. The grid would be divided into 10 X 10 unit boxes. Now the grid is 10 boxes wide and 5 boxes high. These boxes

would then be used to form block paths from the start position to the goal. Figure 5.1 shows how the grid is divided into boxes. Boxes marked with an X illustrate an example block path.

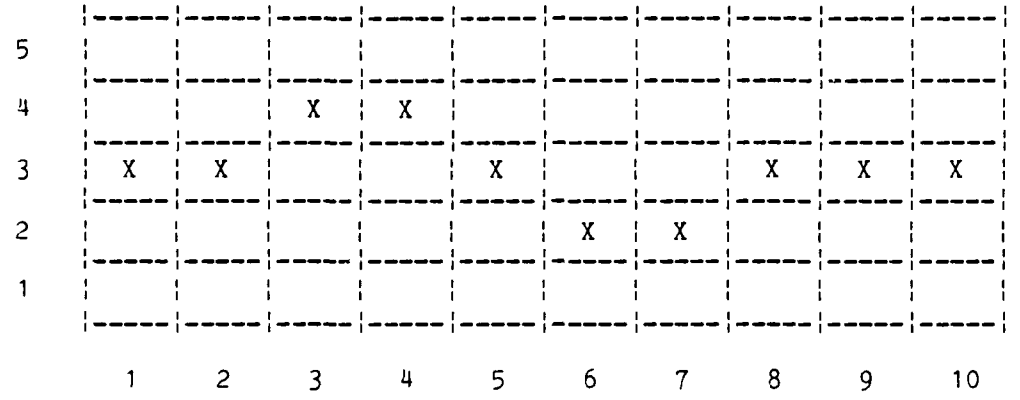


Figure 5.1

Sample Grid and Block Path

Various plans are included in the final version of HPLAN. Although these will be discussed in detail in the next section, they will be discussed briefly now. Different methods for calculating box values in the block paths are used in HPLAN. First, if a threat center is in a box, that box gets the threat value. In addition, any box that has its center included in that threat's circle also receives the threat value. Another method uses the same technique for computation of box values as above, but does so with smaller 5 X 5 boxes. Once these smaller boxes have values, four of them are added together to obtain a value for the big 10 X 10 box that contains them. A third method using small boxes considers only the eight surrounding boxes as possible candidates for the threat value. The major improvement in the final design of HPLAN is the way in which the best block paths are determined. Since the search space is smaller, an exhaustive search is performed to

find all possible block paths. The best block paths are those that contain the least amount of threats, based on the values in the boxes. In the final design of HPLAN, there are two methods for calculating total block path cost. One method simply adds the values for all the boxes in a block path for the block path cost. The other method adds only the values of boxes in its block path that have not aready been added from the same threat. In other words, if a threat covers two adjacent boxes, and both boxes are contained in the block path, the value of only one box will be added for that threat. This eliminates counting the same threat twice in the same block path. Differences between the two methods will be noted later.

After determination of the best block paths, the block path now becomes the new, smaller search space for the problem. Now an exhaustive search is feasible because the search space is much smaller. An exhaustive search is performed within a block path to locate the best flight path. It is at this lower level that details of the original problem can be introduced. Calculating actual flight path costs through random threats, determining the best flight path from beginning to end, and calculating path lengths are details that were ignored at the higher level plans but are now considered. When this is accomplished, the finalized flight path with cost and length computed, is found and the original problem is solved.

VI. HPLAN IMPLEMENTATION

The hierarchical planning algorithm (HPLAN) is implemented in the programming language PASCAL. Program coding and testing were performed on both DEC VAX 11/750 and VAX 11/780 mini-computers. Graphics were designed and produced on the IBM 3083 system using SAS/GRAPHICS and a CALCOMP 1012 plotter. The following sections will describe the basic components of the HPLAN implementation. First, the HPLAN algorithm will be described in general terms to explain the program flow. Following this, block path construction and methods for computing box values will be discussed. The next sections describe block paths and flight paths. The final sections describe and illustrate HPLAN input and output.

HPLAN ALGORITHM

The HPLAN program generates aircraft routes and finds a path through random hostile environments using hierarchical planning. The algorithm begins reading an input data file containing program options and a set of random threats. Program options will be explained throughout the following sections. The set of threats in the input file define the hostile environment the aircraft must fly through. Each threat is defined by the x, y coordinates of its center, a radius to define how far the threat extends, and a value to represent its cost. After the data is read, the program processes the threats. This includes storing threat values and computing individual box values in the grid. Once all the boxes defined in the grid have a value assigned, block paths containing these boxes are generated. All possible block

paths are examined to determine which ones are the best candidates for an exhaustive search to find the best flight path. Two block paths are chosen and an exhaustive search is then performed on both. Each exhaustive search produces the best possible flight path within the block path. The two flight paths are compared and the one that has the least cost will be the final path selected. Should both flight paths incur the same cost, the shorter flight path is considered best. Finally, the best flight path, its block path, program options and threat values are output. The program flow just discussed is shown in Figure 6.1. The entire program listing for HPLAN is presented in Appendix A.

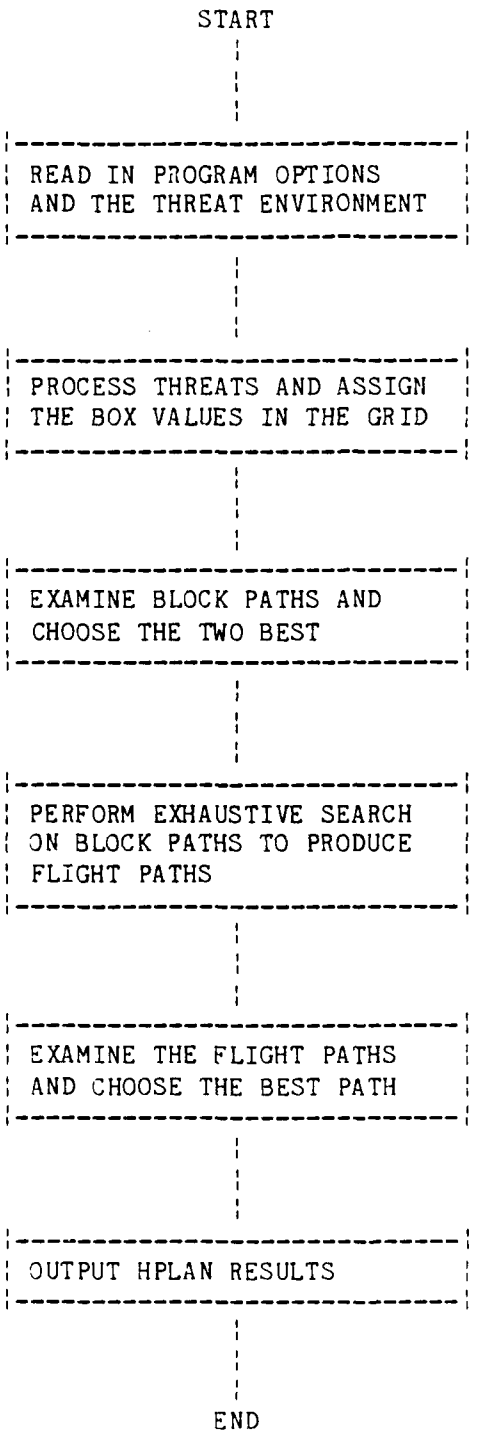


Figure 6.1

Program Flow for HPLAN

```
ENTER TYPE OF BLOCK PATH -- BIG or SMALL

BIG

ENTER COST OPTION -- ALL or DECONFLICT

ALL

ENTER BOX THREAT COVERAGE -- CENTER or TOTAL

TOTAL

ENTER MAXIMUM XY BLOCK COORDINATES -- XMAX YMAX

10 5

ENTER NUMBER OF DIVISIONS AND THE NUMBER OF
POINTS FOR EACH DIVISION -- NUMDIV PTS

10 10

ENTER SEED

31583

ENTER NUMBER OF THREAT CATEGORIES

3

ENTER TOTAL THREAT DENSITY

1.0

FOR THREAT CATEGORY 1 ENTER RADIUS, COST, DENSITY

5 40 0.4

FOR THREAT CATEGORY 2 ENTER RADIUS, COST, DENSITY

7.5 25 0.3

FOR THREAT CATEGORY 3 ENTER RADIUS, COST, DENSITY

10 10 0.3
```

Figure 6.9

Sample Run of THREAT_BLDR

program listing for THREAT_BLDR is presented in Appendix B. A sample of THREAT_BLDR prompts with user inputs underlined is shown in Figure 6.9. An example file produced by THREAT_BLDR and used for input to HPLAN is given in Figure 6.10.

		Cost Option	Box Coverage	X, Y	Num Div	Pts/ Div
Type Path	BIG	ALL/DECONFLICT	TOTAL	10,5	10/20	2-20
		ALL	CENTER/TOTAL	20,10	10/20	2-20
	SMALL	ALL/DECONFLICT	CENTER/TOTAL	20,10	20	2-20

Table 6.1

HPLAN Options

HPLAN INPUT

Program inputs for HPLAN consist of program options and values representing the threat environment. These values are read by HPLAN via a data file. This data file, called DATAIN, is produced by a separate program, THREAT_BLDR. THREAT_BLDR is written in PASCAL and prompts the user for HPLAN options and other information required to produce a random threat environment. The user inputs the type of block path, cost option, box coverage, maximum x and y block coordinates, number of divisions, and the number of points for each division. These inputs are stored in DATAIN and used by HPLAN for developing a specific plan. Table 6.1 lists the options available in HPLAN and their various combinations. The remaining portion of THREAT_BLDR prompts the user for information for developing the threat environment. THREAT_BLDR uses a random number generator to produce x, y coordinates for the individual threats. A random seed, the number of threat categories, and the total threat density are inputs requested. A threat category defines a set of threats with the same radius and cost. The total threat density is the area of the grid that is covered by the threats. In addition to the radius and cost, the user is asked to input the density for each threat category. This defines individual threat category density relating to the total threat density. The sum of the threat category densities is equal to one. After all threats are generated, THREAT_BLDR places an end of file marker in DATAIN to identify the end of input.

The random number generator and portion of code in THREAT_BLDR that creates the random threat environment is the same as that used in [5]. This produces identical threat environments to facilitate comparison between hierarchical planning and heuristic search. The

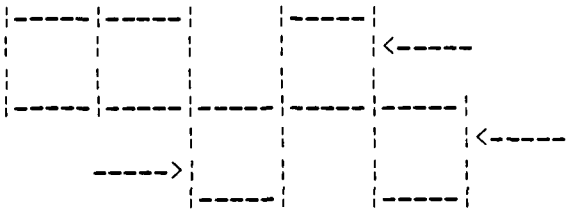


Figure 6.6

Division Lines

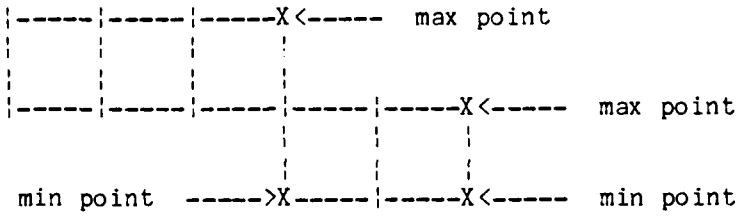


Figure 6.7

Min/Max Points

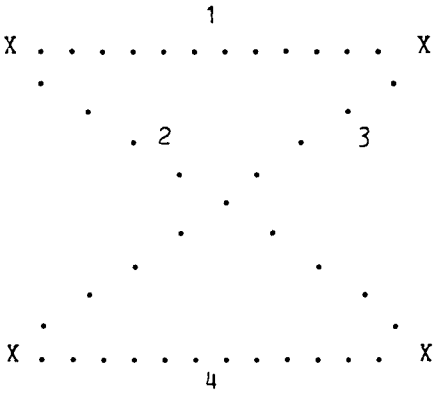


Figure 6.8

Possible Legs

division. The best path to a division is the one with the least cost. If more than one path has the same cost, the shortest path is considered best. This process continues from division to division until the final goal is reached. The best flight path within the block path is then selected.

Cost computation of a flight leg from one point to another involves checking for intersection between a line and a threat circle. If a leg begins outside of a threat and intersects the circle, the threat cost is assigned to the cost of that leg. If a leg begins inside a threat circle, the threat cost is not included in the leg cost because it would have already been assigned to another leg of that path. Legs that are tangent to a threat circle are not considered to be in the threat area and are therefore not assigned the threat cost. A single leg can incur the cost of more than one threat if it intersects several threats. The threat costs are added together to compute the cost of the leg. The total cost for a flight path is the addition of all costs for all the legs in the flight path.

Following the exhaustive search of both block paths, two best flight paths exist, one for each block path. The best flight path is then selected. The best flight path is the path with the lowest cost associated with it. Should both flight paths have the same cost, the shortest flight path is selected.

division. The best path to a division is the one with the least cost. If more than one path has the same cost, the shortest path is considered best. This process continues from division to division until the final goal is reached. The best flight path within the block path is then selected.

Cost computation of a flight leg from one point to another involves checking for intersection between a line and a threat circle. If a leg begins outside of a threat and intersects the circle, the threat cost is assigned to the cost of that leg. If a leg begins inside a threat circle, the threat cost is not included in the leg cost because it would have already been assigned to another leg of that path. Legs that are tangent to a threat circle are not considered to be in the threat area and are therefore not assigned the threat cost. A single leg can incur the cost of more than one threat if it intersects several threats. The threat costs are added together to compute the cost of the leg. The total cost for a flight path is the addition of all costs for all the legs in the flight path.

Following the exhaustive search of both block paths, two best flight paths exist, one for each block path. The best flight path is then selected. The best flight path is the path with the lowest cost associated with it. Should both flight paths have the same cost, the shortest flight path is selected.

FLIGHT PATHS

An exhaustive search performed on the block paths produces the best flight path for each block path. The two options that dictate how the exhaustive search is performed are number of divisions and number of points per division. Divisions are sections along a block path that define the beginning and end of a flight leg. The number of divisions will be either 10 or 20 depending on the type of block path. Either 10 or 20 is available for the BIG option and only 20 will be used for the SMALL option. Each division is divided equally based on the number of points per division. Divisions define the distance in the x direction for the legs of a flight path, while the number of points per division define the y coordinates. Divisions have minimum and maximum y coordinates defined. For each division, this is the minimum and maximum points on a vertical line connecting two boxes in the block path. Division lines and their minimum and maximum points are illustrated with a sample block path in Figures 6.6 and 6.7. The number of points per division defines how many points there are on each division line. The minimum number of points is 2 and the maximum is 20. The legs of a flight path start at one division, at each point on that division, and connect to the next division, at each one of its points. For example, if there are two points per division, the number of possible legs to the next division is four. This is illustrated in Figure 6.8. As each leg is generated from one division to the next, cost and length are computed for that flight leg. After all possible legs are generated and their costs computed, from one division to the next, there exists one best path to each point on the next division. For example, if there are five points per division, there will be exactly five best paths to that

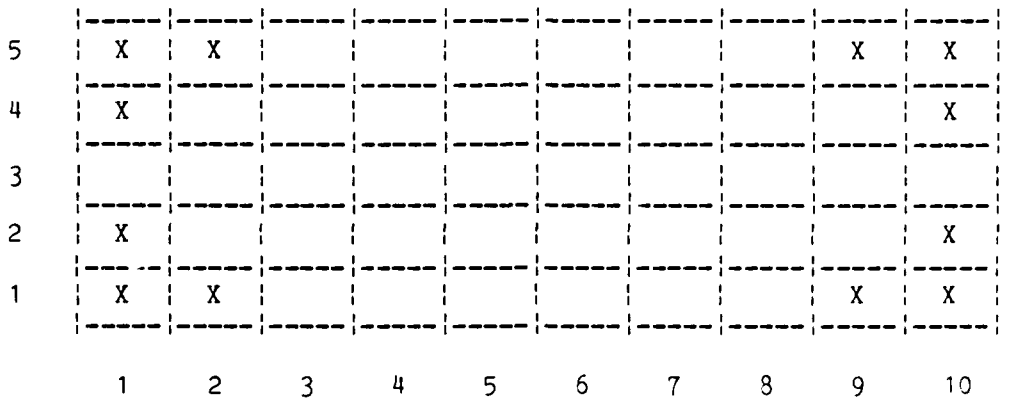


Figure 6.4

Big Block Paths

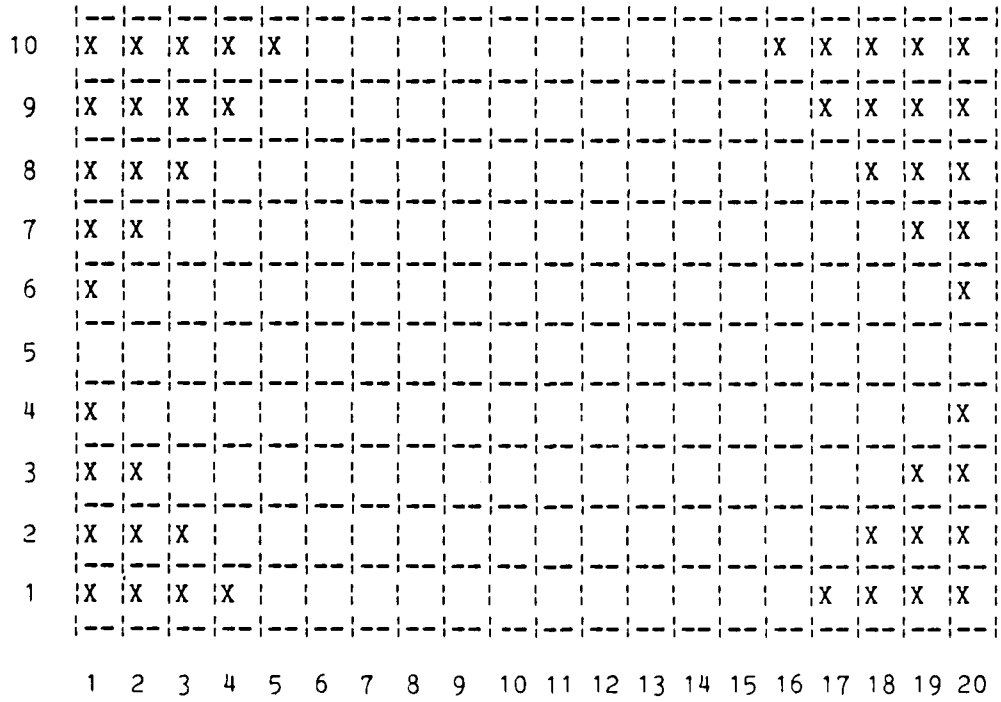


Figure 6.5

Small Block Paths

discussing box values, it was noted that threat values for a box would be stored separately with the DECONFLICT option. If a threat covers two different boxes, then both boxes are assigned the cost of that threat. With the DECONFLICT option, if these two boxes are contained in the same block path, the value from the same threat will only be added to the total score once. This method ensures a threat cost will only be added one time in computing the score for a block path. Both options, ALL or DECONFLICT, are available on every plan except when the block path is BIG and the grid contains small boxes. This is because the small box values are added together to obtain big box values.

After all block paths have been generated and scored, two are selected for an exhaustive search. If there are two block paths that have the best score, then both paths are selected. If more than two block paths are tied with the best score, the two block paths that contain the most boxes exclusive of each other are selected. This is determined by comparing the y box coordinates of each block path with the other block paths that are tied with the best score. If there is only one block path with the best score, it is selected along with a block path that has the next best score. Once two block paths are selected, an exhaustive search is performed on each to produce flight paths.

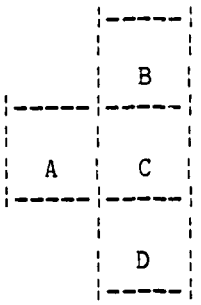


Figure 6.3

Next Box Moves

because it continues the path in a forward direction, but it may not be considered because it cannot possibly lead to the goal box. Figures 6.4 and 6.5 illustrate both BIG and SMALL block paths and indicate with an 'X' those boxes in each grid that are not possible candidates to produce a valid block path. In Figure 6.4 for example, box 9, 5 is not a candidate for a possible block path because only boxes 10, 5 and 10, 4 can be reached from it and they cannot reach the goal box by valid moves. Therefore, when generating the next possible move from box 8, 4, box 9, 5 is not a possible move because the goal box cannot be reached.

As each possible block path is generated, a score is assigned to the block path. Since all boxes now have values assigned to them, each block path can be assigned a score equal to the values in the boxes in the block path. Lower scores indicate better block paths. The method for determining a block path score is given by the cost option, ALL or DECONFLICT. The ALL option calculates the score by adding all the values in the boxes that are contained in the block path. For example, if there are 10 boxes, each with a value of 10, the final score for the block path is 100. The DECONFLICT option also adds values from the boxes, but may not add all the values. In the previous section

into 200 small boxes, but the type of block path is BIG, indicating 10 X 10 boxes, values in the small 5 X 5 boxes are added to obtain values for the big 10 X 10 boxes. Each 10 X 10 box contains four 5 X 5 boxes. The values from these four boxes are added together to obtain the value for the 10 X 10 box. Boxes in the grid now have values to indicate the threat cost assigned to each box.

BLOCK PATHS

The next process after all boxes have values assigned, is to generate block paths and evaluate them to determine the best block paths. The quality of the final solution depends on this process. A block path is a path from the starting point on the grid to the goal point via boxes in the grid. If the path option in HPLAN is BIG, then the block paths consist of 10 X 10 boxes, otherwise 5 X 5 boxes are used. First, all possible block paths in the grid are generated. Possible block paths are those that contain the starting point (0, 25) and continue through adjacent boxes until the block path reaches the box containing the goal point (100, 25). This process begins with the box containing the starting point (0, 25) and uses recursion to build all possible block paths through the entire grid. The next possible box in a block path from the current box are those that continue the path in a forward direction towards the goal box. From a current box, there are at most three boxes for the next move in the block path. This is illustrated in Figure 6.3. If box A is the current box, then the next box in the block path must be B, C, or D.

Since the block path must be "driven" towards the goal box (the box containing the goal point), several boxes in the grid are not candidates for a possible block path. A box may be a valid candidate

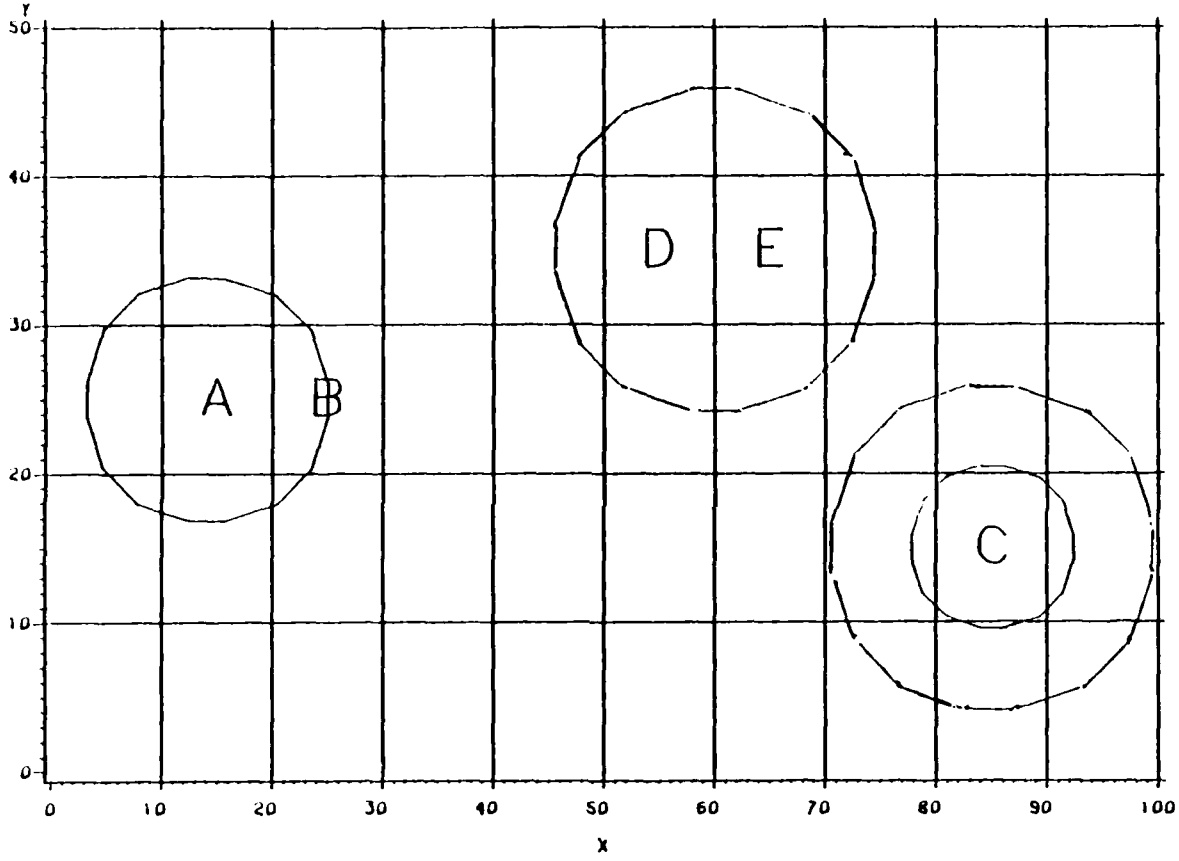


Figure 6.2

Box Values

BOX VALUES

Values assigned to boxes in the grid are calculated differently depending on the HPLAN options selected. After the threat environment is input to HPLAN, each individual threat is processed to determine which box in the grid contains the threat center. This box is then assigned a value equal to that threat's cost value. Since a threat may extend over more than one box, a check is made to determine what other boxes are affected by the same threat. The other boxes checked for containment within the threat are determined by the box coverage option. If the option is TOTAL, then all boxes within the threat radius are considered. If the box coverage option is CENTER, then only those eight boxes that are adjacent to the box containing the threat center are considered. The center option is only available when computing values for small 5 X 5 boxes. In either case, those boxes considered are assigned the threat cost if the threat contains the center point of a box, in which case we say that the box is covered by the threat. Boxes that are covered by more than one threat add the threat costs together to compute the box value. This is illustrated in Figure 6.2. Box A is assigned the threat cost but box B is not because the center of box B is not contained in the threat. Box C is assigned the sum of the two threat costs because the center of box C is contained in both threats. Boxes D and E will be assigned the threat cost because both their centers are contained in the threat. The exception to this is when the cost option DECONFLICT is selected. This method will store the threat costs for a box separately for block path score computation. The reason for storing the values separately will be explained in the next section.

This process continues for each threat. If the grid is divided

BLOCK PATH CONSTRUCTION

HPLAN is developed to allow for several processing options. A set of options defines a plan to produce a flight path through a threat environment. Options are considered throughout the algorithm and dictate several processing decisions in HPLAN. HPLAN options exist for describing how the grid is divided and how block paths are built. The grid size is 100 units wide and 50 units high. To describe how the grid is divided, x and y values are input to define how many boxes the grid will contain. For example, if x is 10 and y is 5, the grid will be 10 boxes along the x-axis and 5 boxes along the y-axis. Therefore, the grid will contain 10 X 10 unit boxes. Similarly, if x is 20 and y is 10, the grid will contain 5 X 5 unit boxes. The size of the boxes in the grid will either be 5 X 5 or 10 X 10 depending on the x and y values.

To define how block paths are built, the type of path is input. If the type of path is BIG, block paths will be constructed with 10 X 10 boxes. If the type of path is SMALL, the block paths will consist of 5 X 5 boxes. The x and y values and the type of path together define three plans for block path construction depending on the combinations selected. If the type of path selected is BIG, then x and y may be either 10,5 or 20,10 respectively. The BIG/10,5 combination indicates the final block paths are 10 X 10 boxes with box values computed based on the 10 X 10 boxes. The BIG/20,10 combination again defines the final block path as 10 X 10 boxes, but box values are first computed using the 5 X 5 boxes and are then added to obtain values for the 10 X 10 boxes. The third plan these combinations produce is the SMALL/20,10 option. This defines block paths with 5 X 5 boxes with box values computed using 5 X 5 boxes.

```
BIG
ALL
TOTAL
10 5
10 10
54.16   1.28   40   5.0
36.29  31.99   40   5.0
22.45  35.79   40   5.0
54.78  78.49   40   5.0
27.58  39.54   25   7.5
38.90  23.65   25   7.5
12.11  97.23   25   7.5
43.21  45.98   25   7.5
23.67  84.13   25   7.5
19.78  21.56   10  10.0
61.23  39.33   10  10.0
 3.01  30.11   10  10.0
   -1     -1   -1    -1
```

Figure 6.10

Sample DATAIN File

HPLAN OUTPUT

HPLAN output results reflect the input options and threat environment, a matrix representing values for boxes in the grid, final block and flight paths, and selected statistics. Figure 6.11 presents a sample output listing from HPLAN. The first section lists all the HPLAN options selected and the input threat environment. The input threat environment consists of the total number of threats and the x, y coordinates, radius, and cost for each threat. A matrix is also output containing the values assigned to the boxes in the grid. If the cost option is ALL, the final values for each box are shown. If the path option is BIG with x, y coordinates 20,10, the matrix with small box values and the big box values with the small boxes added, are both shown. If the cost option is DECONFLICT, the matrix output will show the individual threat values assigned to each box. After all box values are displayed, the block path that produced the final flight path is given with its x, y coordinates and score. The flight path is output with x, y coordinates, the cost of the path, and the path length. The CPU time in milliseconds is also shown. This represents the CPU time used not including data input and output. Graphic output showing the threat environment and flight path is produced using information from the HPLAN output.

HPLAN RESULTS:

OPTIONS SELECTED:

```
TYPE OF BLOCK PATHS --  BIG
BLOCK PATH COSTS    --  ALL
BOX THREAT COVERAGE --  TOTAL
XY BLOCK LIMITS     --  10  5
NUMBER OF DIVISIONS --  10
POINTS PER DIVISION --   9
```

INPUT THREAT ENVIRONMENT:

TOTAL NUMBER OF THREATS -- 18

THREAT INPUTS -- X Y RADIUS COST

1.92	40.26	5.0	50
90.13	31.94	5.0	50
80.31	13.91	5.0	50
83.21	42.73	5.0	50
11.14	41.33	5.0	50
65.46	17.82	5.0	50
24.10	34.22	5.0	50
29.05	25.19	5.0	50
73.87	16.81	5.0	50
91.18	45.30	5.0	50
40.15	25.81	5.0	50
94.03	21.13	5.0	50
27.62	10.46	7.5	25
4.79	8.14	7.5	25
65.97	32.45	7.5	25
15.64	24.30	7.5	25
9.88	16.02	10.0	10
43.79	0.08	10.0	10

Figure 6.11

Sample Output Listing from HPLAN

```
BOX THREAT VALUES

  5  0    50     0     0     0     0     0     0    50    50
  4 50     0    50     0     0     0    25     0    50     0
  3  0    25    50    50    50     0     0     0     0    50
  2 35    10    25     0     0     0    50   100    50     0
  1 25     0    25     0    10     0     0     0     0     0
     1     2     3     4     5     6     7     8     9    10

BLOCK PATH

          X          Y

          1          3
          2          4
          3          5
          4          5
          5          5
          6          5
          7          5
          8          4
          9          3
         10          3

SCORE =   50


FLIGHT PATH

          X          Y

       0.00      25.00
      10.00      32.50
      20.00      40.00
      30.00      40.00
      40.00      40.00
      50.00      40.00
      60.00      40.00
      70.00      40.00
      80.00      35.00
      90.00      30.00
     100.00      25.00

COST =  50.00

LENGTH = 108.54

CPU TIME(msec) =       17860
```

Figure 6.11 -- concluded

Sample Output Listing from HPLAN

VII. HPLAN RESULTS

Test results from HPLAN are presented for eight test cases. Each test case represents a different threat environment. The random threat environments used here are the same as those used in [5] to facilitate comparison between hierarchical planning and heuristic search. Results from each test are presented separately. First, a discussion of a test result is given. This includes comments concerning the different run options, discussion of block paths, flight paths, and comparison to heuristic search results. Next, the results are presented in tables. Each table will describe the threat environment which includes the random seed used to create the threats, the total threat density, the number of threats generated, and the number of threat types. For each threat type, its radius, cost, and density are given. A table entry represents a computer run and consists of the type of path, cost option, box coverage, x and y values, number of divisions, and the number of points per division. The last three columns in a table entry give the CPU time in seconds, the path cost, and the path length. The last three items are also given for the heuristic search method. Multiple runs were made in [5] on test cases and this entry represents the best path results. Finally, graphs are presented for test cases. Graphs illustrate the grid, threat environment, and the flight path.

Various program options were used on the test cases. Only two test cases (tests 2 and 3) used the type path SMALL option because of CPU time requirements. Processing time using the small block paths used

appoximately 54 hours of CPU time. In contrast, CPU time using big block paths never exceeded 8 minutes and was as little as 9 seconds. Tests 3 through 8 contain more run options because these were considered final test cases for the heuristic search results. All possible combinations of HPLAN options are tested against the last five test cases. The number of points per division used for each option are 5, 9, and 17. These were selected because they offer a varying range between the minimum and maximum points, 2 and 20. They were also selected because they successively add points exactly between each other thereby producing results at least as good as the set preceeding them.

TEST 1

The first observation from Table 7.1 is that the x, y values 20, 10 produced better results than 10, 5. Adding values from small boxes to obtain big box values works better in this case. The block paths from the two methods were different. As can be seen in Figure 7.1, the block path forced the flight path for the 10, 5 option to the top portion of the grid. Once in that area, the only way the path can reach the goal is to pass through a threat with a cost of either 25 or 40. Because threats are polygons rather than circles in the graphs, a path may seem to intersect a threat edge when in fact the path actually misses the threat. This can be resolved by checking the path cost that is listed with each graph. The block path with x and y values at 20, 10 forced the flight path to the lower part of the grid. Figure 7.2 shows that the path can reach the goal by passing through just two threats, each having a cost of only 10. Table 7.1 also shows the cost options, ALL and DECONFLICT, when x, y are 10, 5, produce the same results. Additionally, when x, y are 20, 10, the box coverage options CENTER and

TOTAL produce the same results. Initially, it was thought that these different options would produce different block paths, therefore, these observations are surprising. However, later results with different threat environments show that these options will at times produce different block paths. The result using heuristic search method produced the same cost, but was a longer path.

THREAT ENVIRONMENT:

RANDOM SEED: 31583
TOTAL THREAT DENSITY: 1.0
NUMBER OF THREATS: 37
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	40	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	17	102	45	132.10
BIG	ALL	TOTAL	10	5	20	17	226	45	122.38
BIG	DECON	TOTAL	10	5	10	17	106	45	132.10
BIG	DECON	TOTAL	10	5	20	17	227	45	122.38
BIG	ALL	TOTAL	20	10	10	17	102	20	105.79
BIG	ALL	TOTAL	20	10	20	17	222	20	108.83
BIG	ALL	CENTER	20	10	10	17	106	20	105.79
BIG	ALL	CENTER	20	10	20	17	225	20	108.83
HEURISTIC SEARCH RESULTS							12	20	113.80

Table 7.1

Results for Test 1

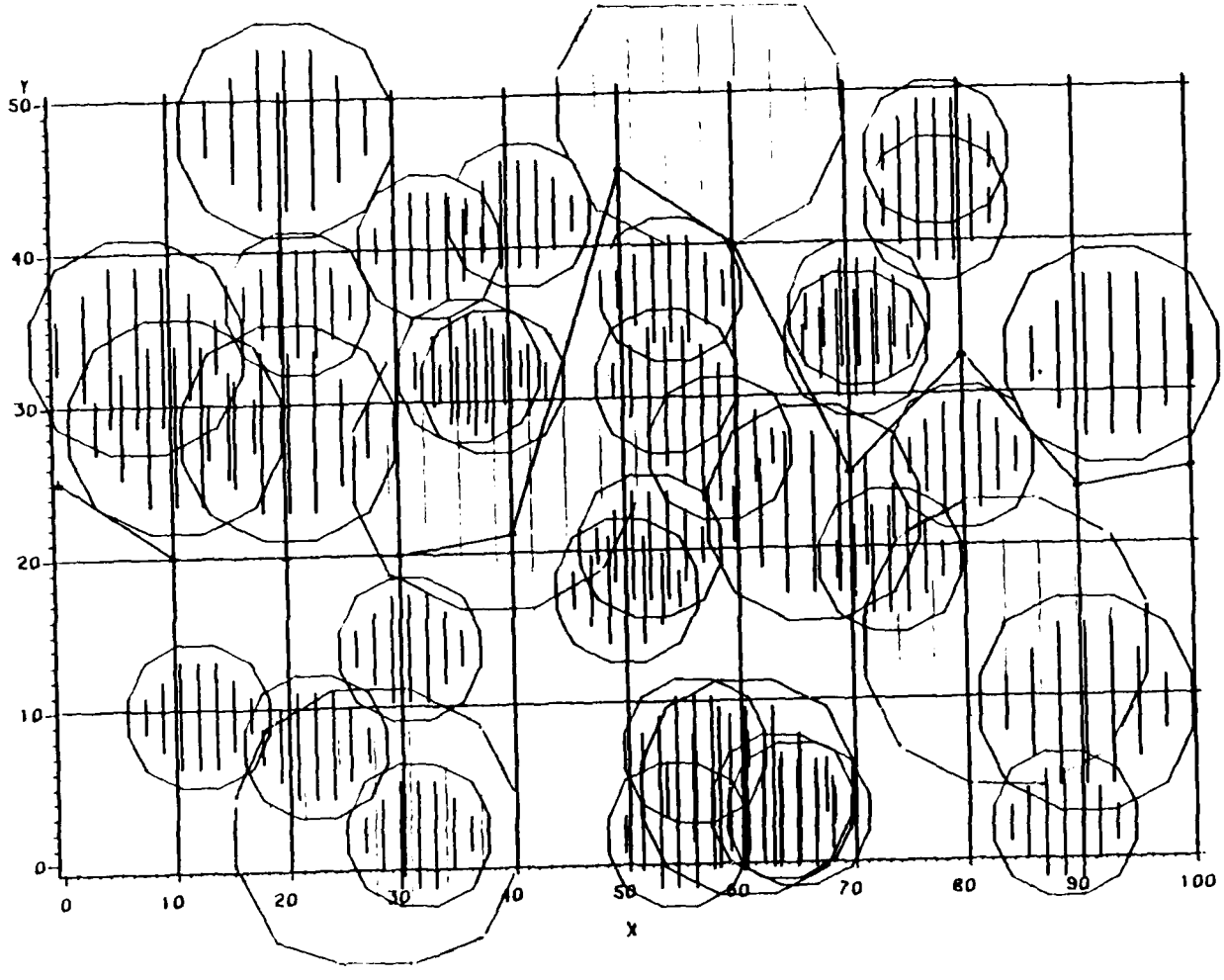


Figure 7.1

Path from Test 1 - 10, 5 Option

Path Cost = 45

Path Length = 132.10

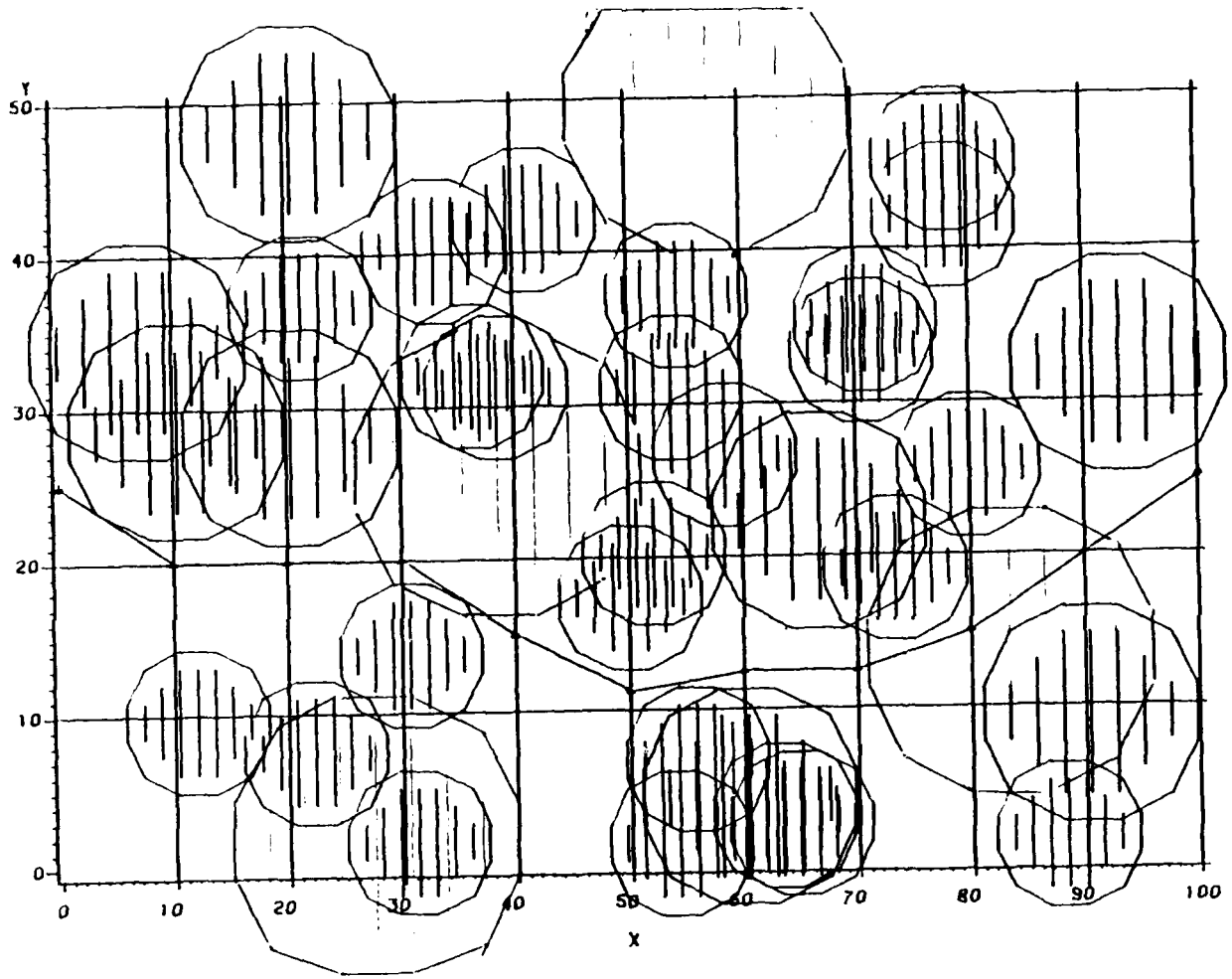


Figure 7.2

Path from Test 1 - 20, 10 Option

Path Cost = 20

Path Length = 105.79

TEST 2

The results for this test show that the x, y values produced the same paths in terms of cost. As with the previous test case, the options ALL/DECONFLICT and CENTER/TOTAL produced similar results. The block paths for the options varied in the first part of the path, but all block paths forced the flight path to reach the goal from the top part of the grid. This produced poor results because the flight path had to pass through at least one of the three high cost threats near the goal. This is illustrated in Figure 7.3. Additionally, all the block paths forced the flight path to intersect the first threat that is closest to the starting point. Therefore, all paths had a cost of 25 assigned after the first leg of the path.

This was the first case in which the type path SMALL option was used. This method produced a block path that moved along the lower section of the grid. Figure 7.4 shows the flight path produced with this option. The path does a good job winding around high cost threats to end up with a path cost of only 20. This method equalled the cost of the path produced by heuristic search. Also, the path using the SMALL option was shorter. The obvious drawback to this method is CPU time. As noted in the beginning of this chapter, the SMALL option is not desirable in terms of CPU time. This was the first test case where the number of points per division made a difference in the cost of the path and not just an improvement in length. With 5 points per division, the SMALL option path had a cost of 45. There were not enough points on a division line for the path to maneuver between threats. When the number of points per division increased to 9 and 17, the path was able to maneuver around and between threats.

THREAT ENVIRONMENT:

RANDOM SEED: 46137
TOTAL THREAT DENSITY: 0.75
NUMBER OF THREATS: 28
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	40	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	17	78	65	109.31
BIG	ALL	TOTAL	10	5	20	17	171	65	112.17
BIG	DECON	TOTAL	10	5	10	17	80	65	109.31
BIG	DECON	TOTAL	10	5	20	17	172	65	112.17
BIG	ALL	TOTAL	20	10	10	17	79	65	109.31
BIG	ALL	TOTAL	20	10	20	17	169	65	111.31
BIG	ALL	CENTER	20	10	10	17	78	65	109.31
BIG	ALL	CENTER	20	10	20	17	175	65	111.31
SMALL	ALL	TOTAL	20	10	20	5	54(HRS)	45	111.18
SMALL	ALL	TOTAL	20	10	20	9	54(HRS)	20	113.50
SMALL	ALL	TOTAL	20	10	20	17	54(HRS)	20	111.88
HEURISTIC SEARCH RESULTS							9185	20	114.96

Table 7.2

Results for Test 2

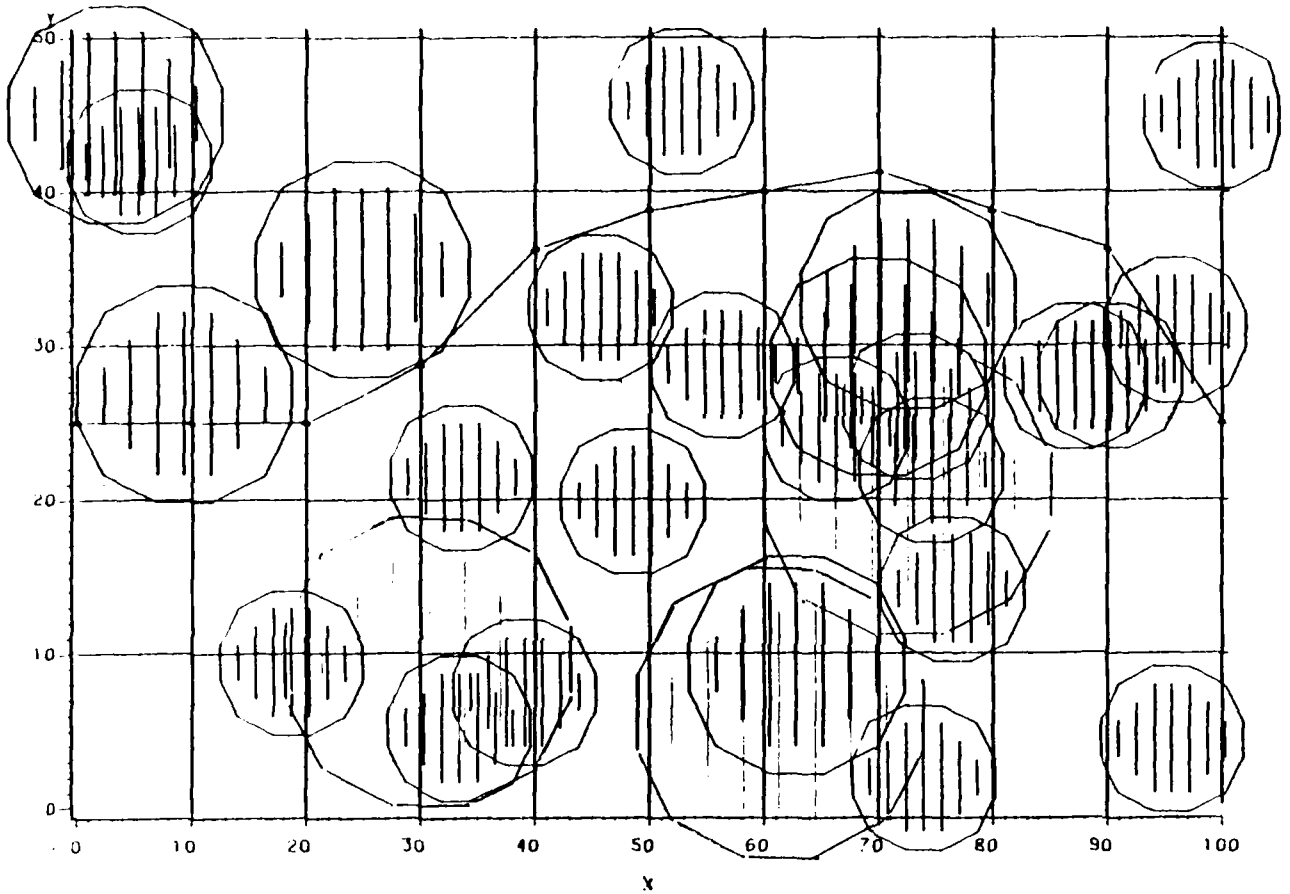


Figure 7.3

Path from Test 2 - BIG Option

Path Cost = 65

Path Length = 109.31

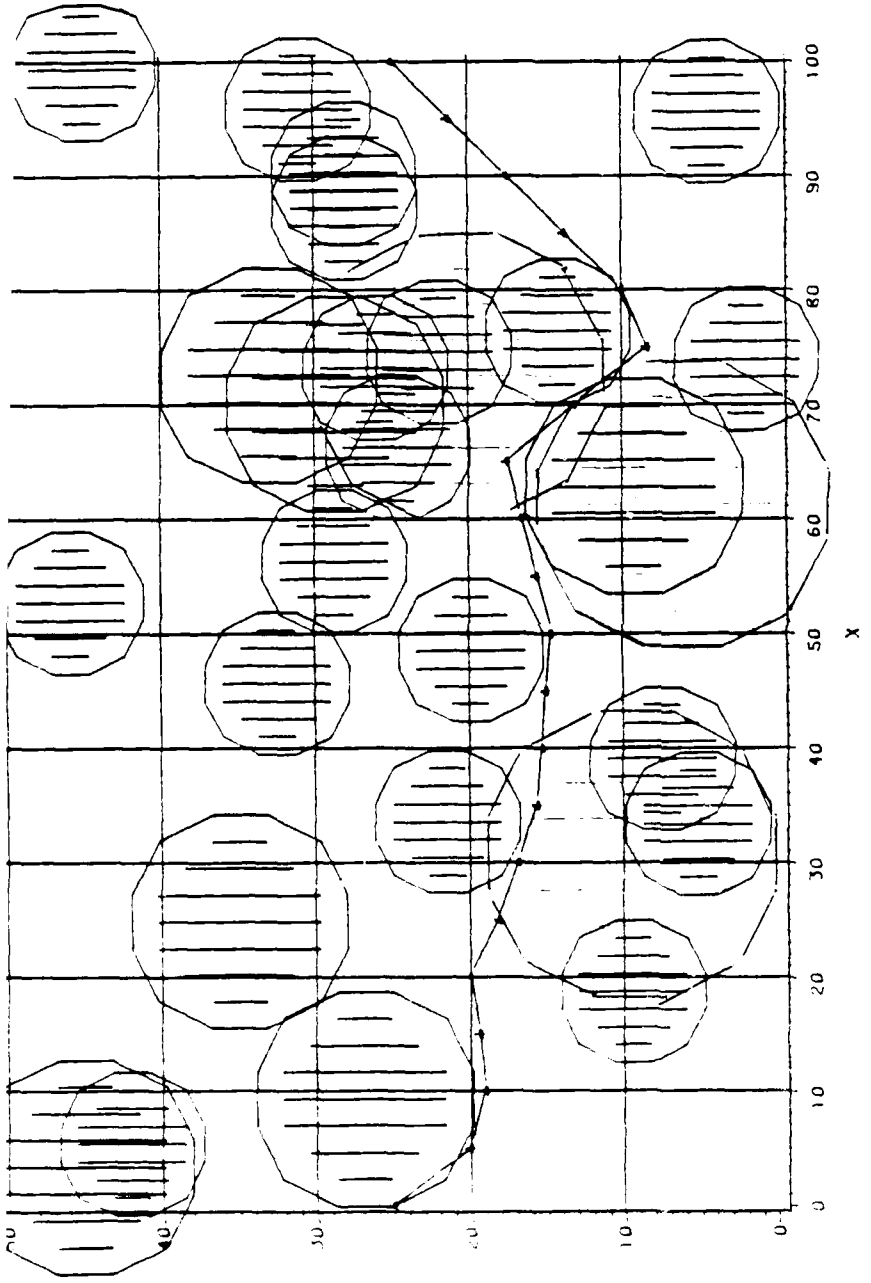


Figure 7.4

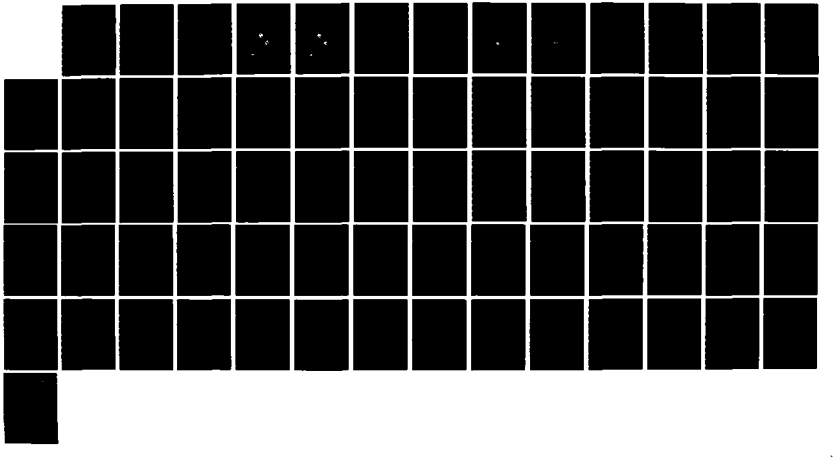
Path from Test 2 - SMALL Option

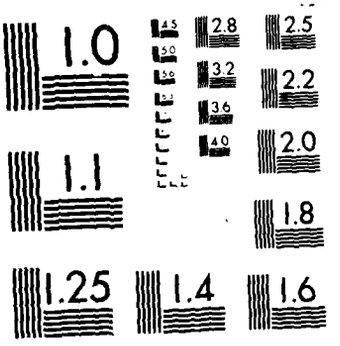
Path Cost = 20

Path Length = 111.88

AD-A156 905 GENERATION OF FLIGHT PATHS USING HIERARCHICAL PLANNING 3/3
(U) AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH
K B KLINE 1985 AFIT/CI/NR-85-37T

UNCLASSIFIED F/G 12/2 NL





TEST 3

This test case had the most dense threat environment consisting of 75 threats. Using 10, 5 for the x, y values produced better results than 20, 10 this time. All options for 10, 5 resulted in the same block path. The best path produced for this test case is shown in Figure 7.5. From the results of this test case, it was observed the difference the number of divisions make on the path. With x, y values at 20, 10, 10 divisions produced much better results than 20 divisions. This was because the flight path with 10 divisions was along the top border of the grid while the path using 20 divisions intersected threats lower in the grid. This was the only other time that the type path SMALL was used. This option produced a path cost of 230. In this case, small block paths did not result in a better path.

The heuristic search results were better in this test case. The best path hierarchical planning could produce was one that cost 225. The best heuristic search path had a cost of 165. This path maneuvered through the threat environment avoiding many of the high cost threats. To obtain this path, the heuristic search took 1650 seconds of CPU time. In this case, however, the CPU expense is reasonable in order to obtain the path cost of 165. This path is illustrated in Figure 7.6.

THREAT ENVIRONMENT:

RANDOM SEED: 143954
TOTAL THREAT DENSITY: 2.0
NUMBER OF THREATS: 75
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	22	325	107.41
BIG	ALL	TOTAL	10	5	10	9	59	275	106.40
BIG	ALL	TOTAL	10	5	10	17	198	275	105.54
BIG	ALL	TOTAL	10	5	20	5	43	275	110.90
BIG	ALL	TOTAL	10	5	20	9	126	225	107.56
BIG	ALL	TOTAL	10	5	20	17	437	225	106.49
BIG	DECON	TOTAL	10	5	10	5	25	325	107.41
BIG	DECON	TOTAL	10	5	10	9	62	275	106.40
BIG	DECON	TOTAL	10	5	10	17	201	275	105.54
BIG	DECON	TOTAL	10	5	20	5	46	275	110.90
BIG	DECON	TOTAL	10	5	20	9	128	225	107.56
BIG	DECON	TOTAL	10	5	20	17	456	225	106.49
BIG	ALL	TOTAL	20	10	10	5	22	285	121.84
BIG	ALL	TOTAL	20	10	10	9	59	285	116.90
BIG	ALL	TOTAL	20	10	10	17	207	285	115.30
BIG	ALL	TOTAL	20	10	20	5	42	380	114.82
BIG	ALL	TOTAL	20	10	20	9	125	355	110.68
BIG	ALL	TOTAL	20	10	20	17	456	345	116.41
BIG	ALL	CENTER	20	10	10	5	22	285	121.84
BIG	ALL	CENTER	20	10	10	9	59	285	116.90
BIG	ALL	CENTER	20	10	10	17	196	285	115.30
BIG	ALL	CENTER	20	10	20	5	42	380	114.82
BIG	ALL	CENTER	20	10	20	9	126	355	110.68
BIG	ALL	CENTER	20	10	20	17	438	345	116.41
SMALL	ALL	TOTAL	20	10	20	17	54(HRS)	230	112.01
HEURISTIC SEARCH RESULTS							1650	165	112.58

Table 7.3

Results for Test 3

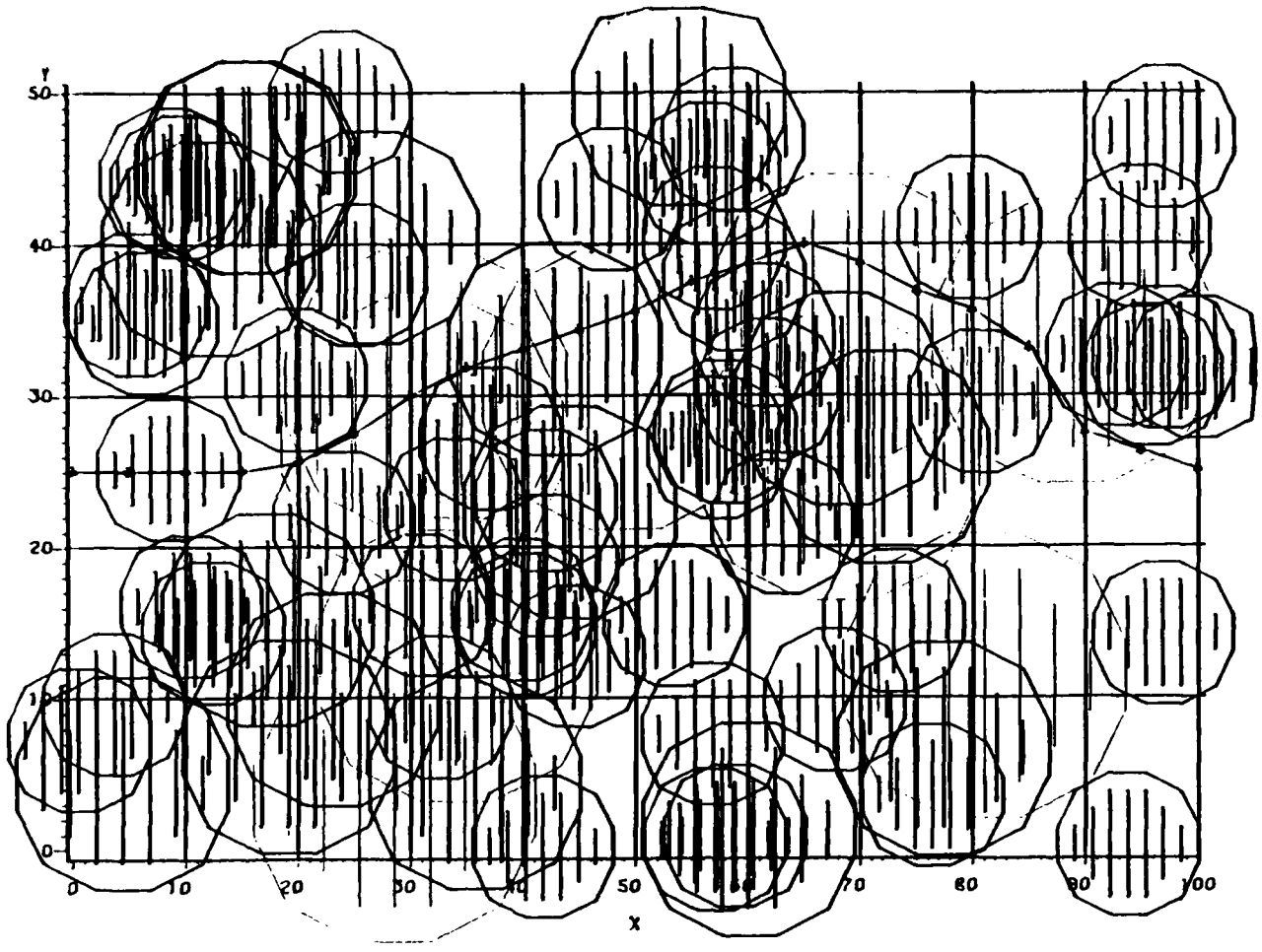


Figure 7.5

Path from Test 3 - 10, 5 Option

Path Cost = 225

Path Length = 106.49

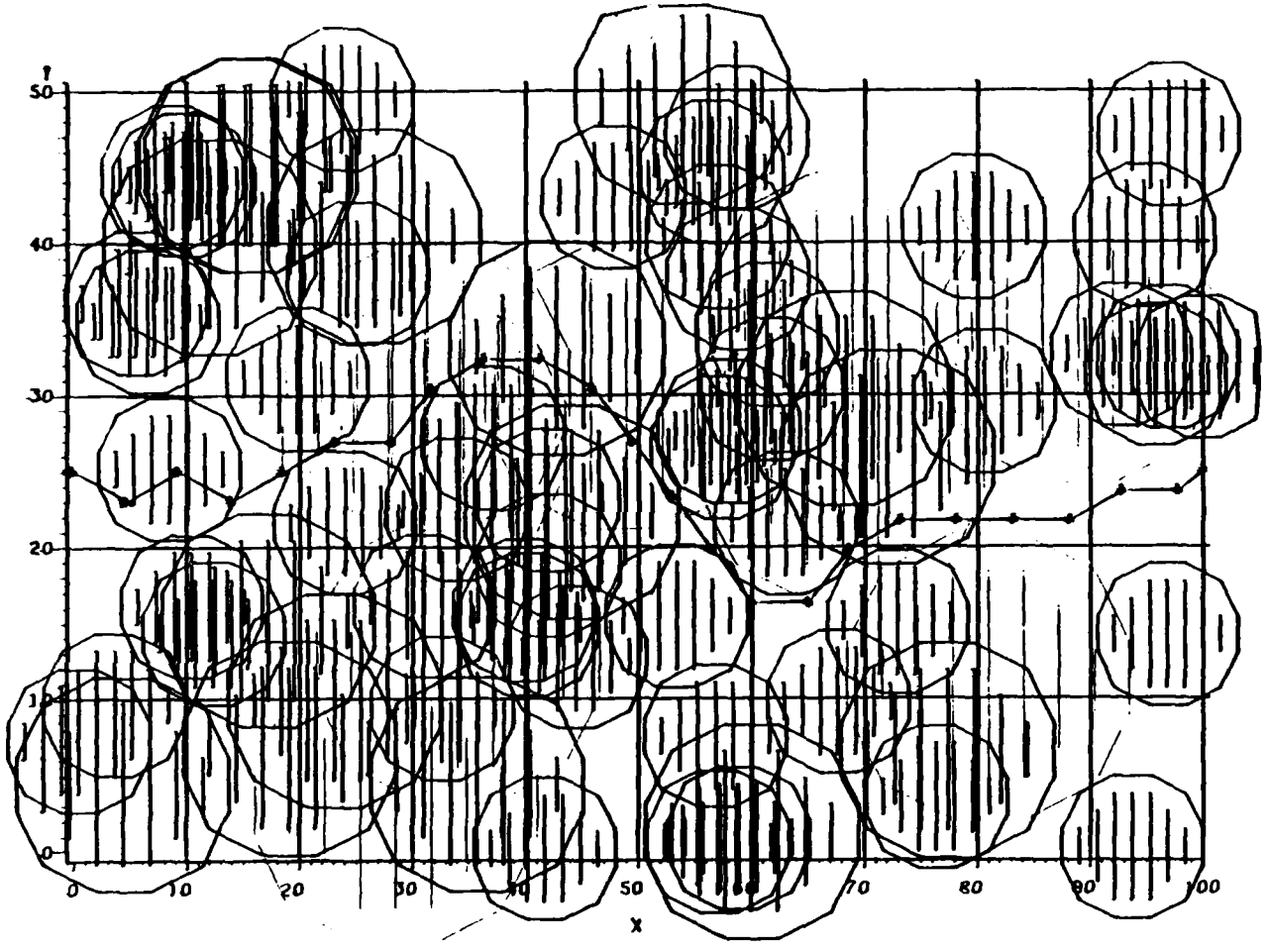


Figure 7.6

Path from Test 3 - Heuristic Search

Path Cost = 165

Path Length = 112.58

TEST 4

This was the first test case in which the DECONFLICT cost option differed from the ALL option. The DECONFLICT option produced much better results than any other method. This was because the block path produced a flight path that could reach the goal from the upper part of the grid rather than from the side or below. This path only intersected one small threat near the goal, while the best path using heuristic search intersected two small threats near the goal because the path came from the bottom part of the grid. The results for the box coverage options TOTAL and CENTER with x, y values 20, 10 were the same again. The best path for this test case was the DECONFLICT cost option using 10 divisions and 17 points per division. This path is shown in Figure 7.7. Figure 7.8 shows the best path using heuristic search.

THREAT ENVIRONMENT:

RANDOM SEED: 21738
TOTAL THREAT DENSITY: 1.5
NUMBER OF THREATS: 57
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	19	220	117.51
BIG	ALL	TOTAL	10	5	10	9	46	205	114.57
BIG	ALL	TOTAL	10	5	10	17	152	170	116.56
BIG	ALL	TOTAL	10	5	20	5	33	280	132.96
BIG	ALL	TOTAL	10	5	20	9	97	280	125.44
BIG	ALL	TOTAL	10	5	20	17	332	230	123.20
BIG	DECON	TOTAL	10	5	10	5	21	205	116.33
BIG	DECON	TOTAL	10	5	10	9	50	205	114.57
BIG	DECON	TOTAL	10	5	10	17	155	155	117.19
BIG	DECON	TOTAL	10	5	20	5	37	205	130.00
BIG	DECON	TOTAL	10	5	20	9	105	205	124.25
BIG	DECON	TOTAL	10	5	20	17	339	155	122.09
BIG	ALL	TOTAL	20	10	10	5	18	255	116.62
BIG	ALL	TOTAL	20	10	10	9	46	230	112.06
BIG	ALL	TOTAL	20	10	10	17	153	230	111.71
BIG	ALL	TOTAL	20	10	20	5	34	230	117.78
BIG	ALL	TOTAL	20	10	20	9	99	230	112.20
BIG	ALL	TOTAL	20	10	20	17	332	205	113.68
BIG	ALL	CENTER	20	10	10	5	18	255	116.62
BIG	ALL	CENTER	20	10	10	9	46	230	112.06
BIG	ALL	CENTER	20	10	10	17	153	230	111.71
BIG	ALL	CENTER	20	10	20	5	33	230	117.78
BIG	ALL	CENTER	20	10	20	9	98	230	112.20
BIG	ALL	CENTER	20	10	20	17	345	205	113.68
HEURISTIC SEARCH RESULTS							4102	210	111.91

Table 7.4

Results for Test 4

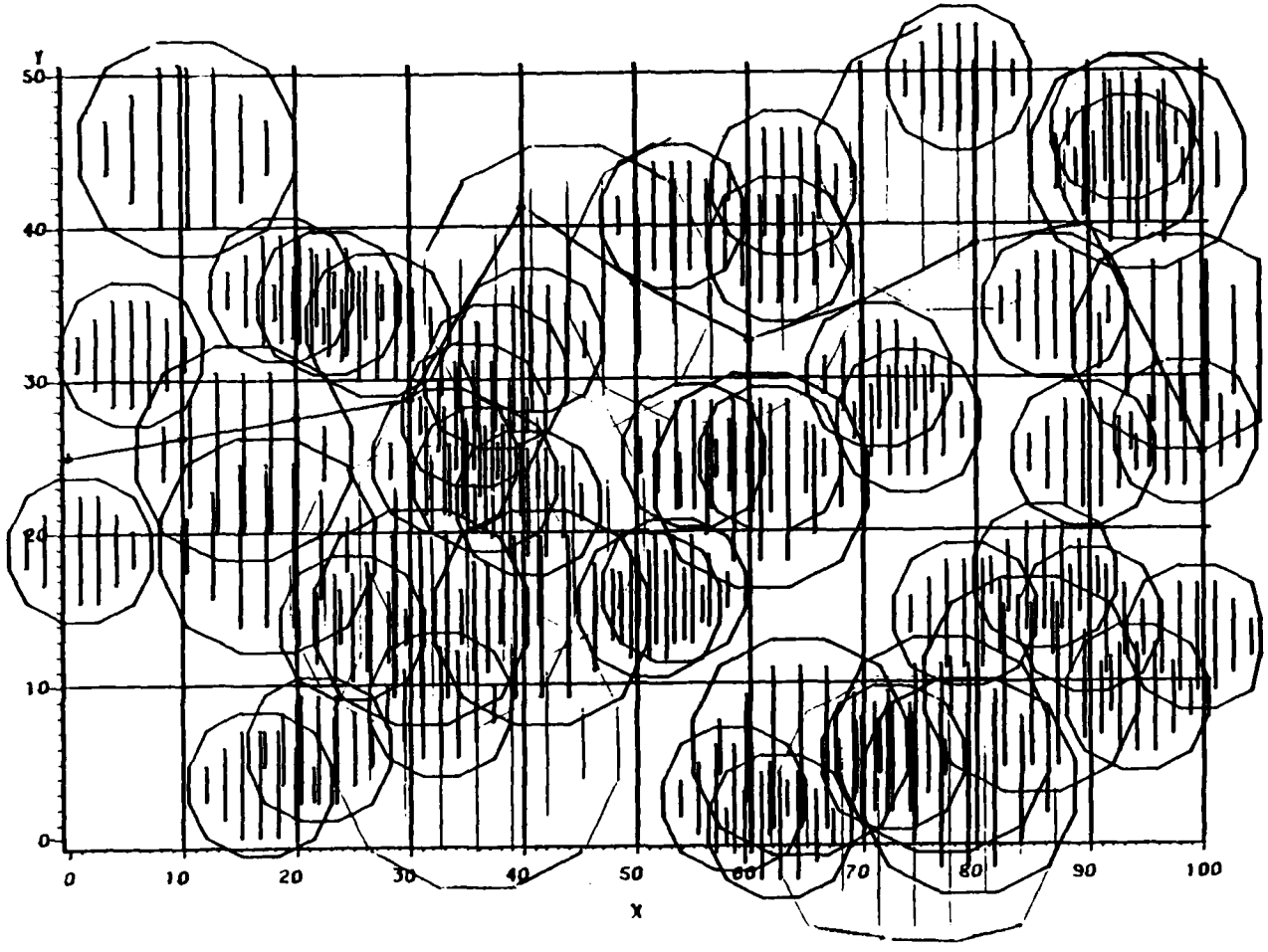


Figure 7.7

Path from Test 4 - DECONFLICT Option

Path Cost = 155

Path Length = 117.19

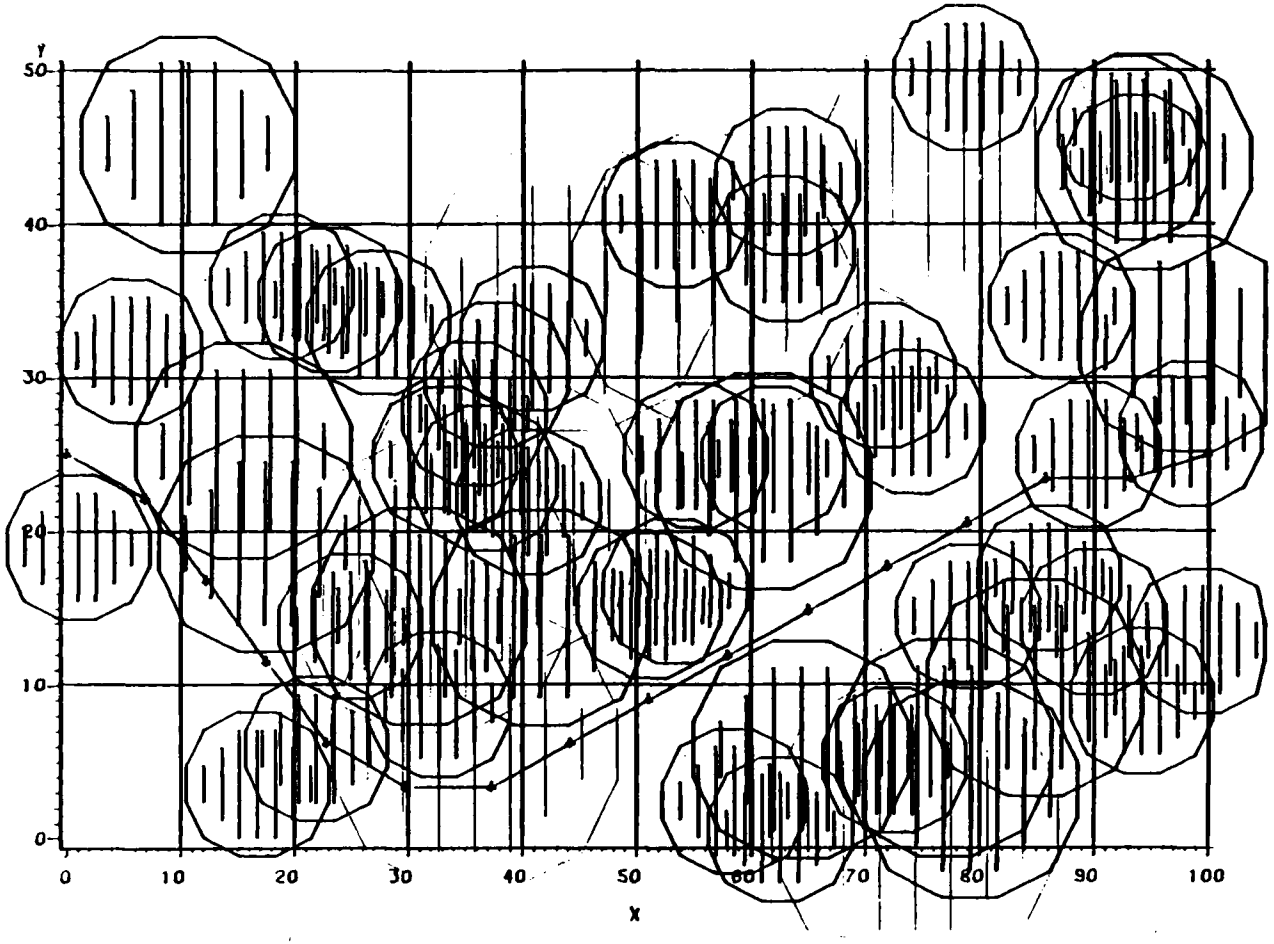


Figure 7.8

Path from Test 4 - Heuristic Search

Path Cost = 210

Path Length = 111.91

TEST 5

The results using x, y values at 10, 5 were better than 20, 10. Also with 10, 5, the ALL and DECONFLICT cost options were different. However, where DECONFLICT was better in test 4, the ALL option was better this time. The block path for the ALL option produced the shortest flight path with the least cost of 105. In the ALL option with x, y at 10, 5, the number of divisions had a significant effect on the path cost in this test. Even though the block paths were the same, the number of divisions forced the flight path to be different. The difference was that when the number of divisions was 20, the path intersected one more threat than it did with 10 divisions. The additional threat contacted had a cost of 50. Both these paths are shown in Figures 7.9 and 7.10. The cost for a path using heuristic search was also 105, however, the hierarchical planning method produced a shorter flight path. The paths from both methods are similar and intersect the same threats.

THREAT ENVIRONMENT:

RANDOM SEED: 89204
TOTAL THREAT DENSITY: 1.0
NUMBER OF THREATS: 37
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	13	105	108.86
BIG	ALL	TOTAL	10	5	10	9	32	105	104.62
BIG	ALL	TOTAL	10	5	10	17	103	105	104.30
BIG	ALL	TOTAL	10	5	20	5	24	155	114.79
BIG	ALL	TOTAL	10	5	20	9	65	155	113.23
BIG	ALL	TOTAL	10	5	20	17	228	155	111.33
BIG	DECON	TOTAL	10	5	10	5	16	105	119.32
BIG	DECON	TOTAL	10	5	10	9	35	105	115.99
BIG	DECON	TOTAL	10	5	10	17	105	105	114.96
BIG	DECON	TOTAL	10	5	20	5	28	105	121.46
BIG	DECON	TOTAL	10	5	20	9	69	105	119.54
BIG	DECON	TOTAL	10	5	20	17	228	105	116.01
BIG	ALL	TOTAL	20	10	10	5	13	110	116.33
BIG	ALL	TOTAL	20	10	10	9	32	110	114.33
BIG	ALL	TOTAL	20	10	10	17	101	110	113.61
BIG	ALL	TOTAL	20	10	20	5	24	160	113.30
BIG	ALL	TOTAL	20	10	20	9	67	110	114.23
BIG	ALL	TOTAL	20	10	20	17	220	110	113.67
BIG	ALL	CENTER	20	10	10	5	13	110	116.33
BIG	ALL	CENTER	20	10	10	9	32	110	114.33
BIG	ALL	CENTER	20	10	10	17	100	110	113.61
BIG	ALL	CENTER	20	10	20	5	23	160	113.30
BIG	ALL	CENTER	20	10	20	9	65	110	114.23
BIG	ALL	CENTER	20	10	20	17	220	110	113.67
HEURISTIC SEARCH RESULTS							1933	105	106.82

Table 7.5

Results for Test 5

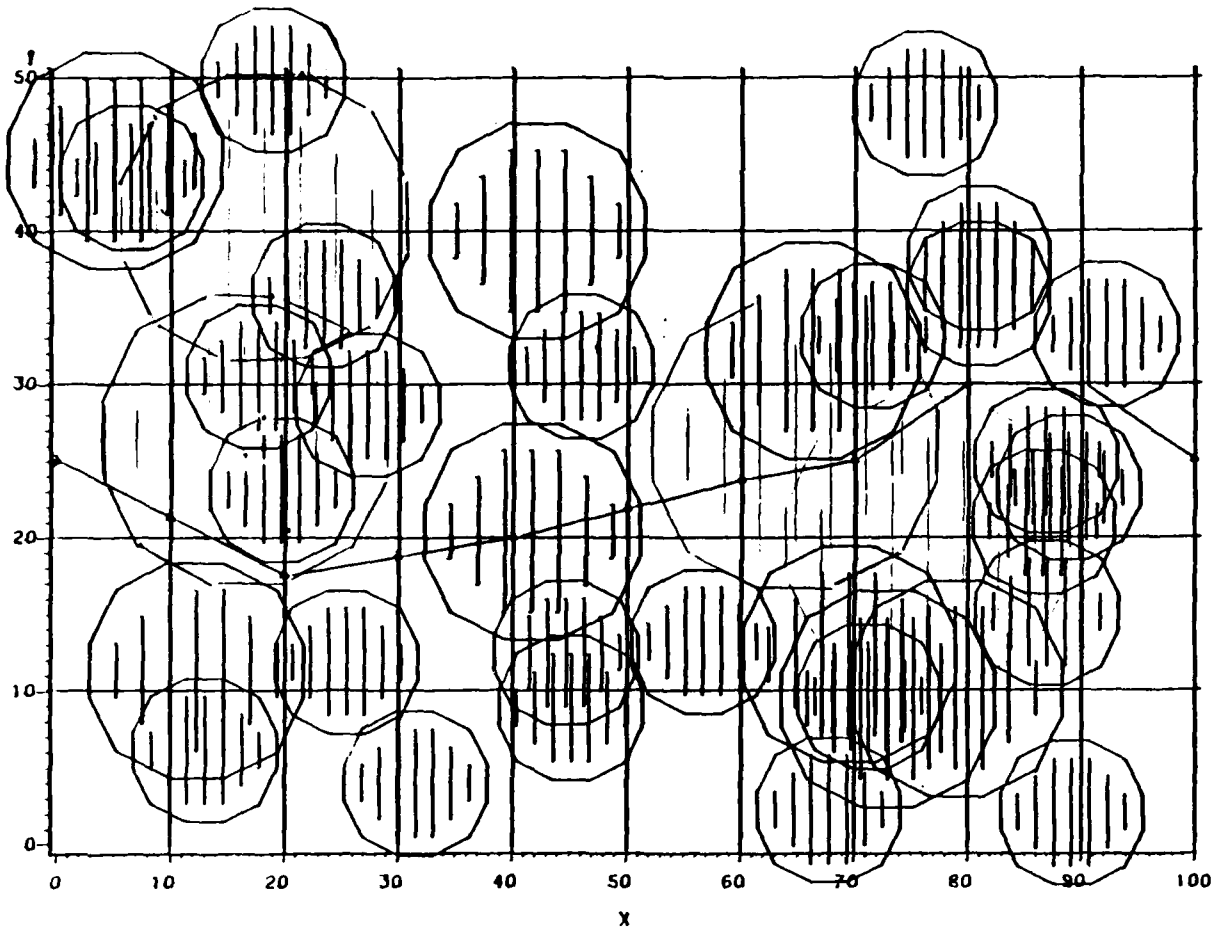


Figure 7.9

Path from Test 5 - 10 Divisions

Path Cost = 105

Path Length = 104.30

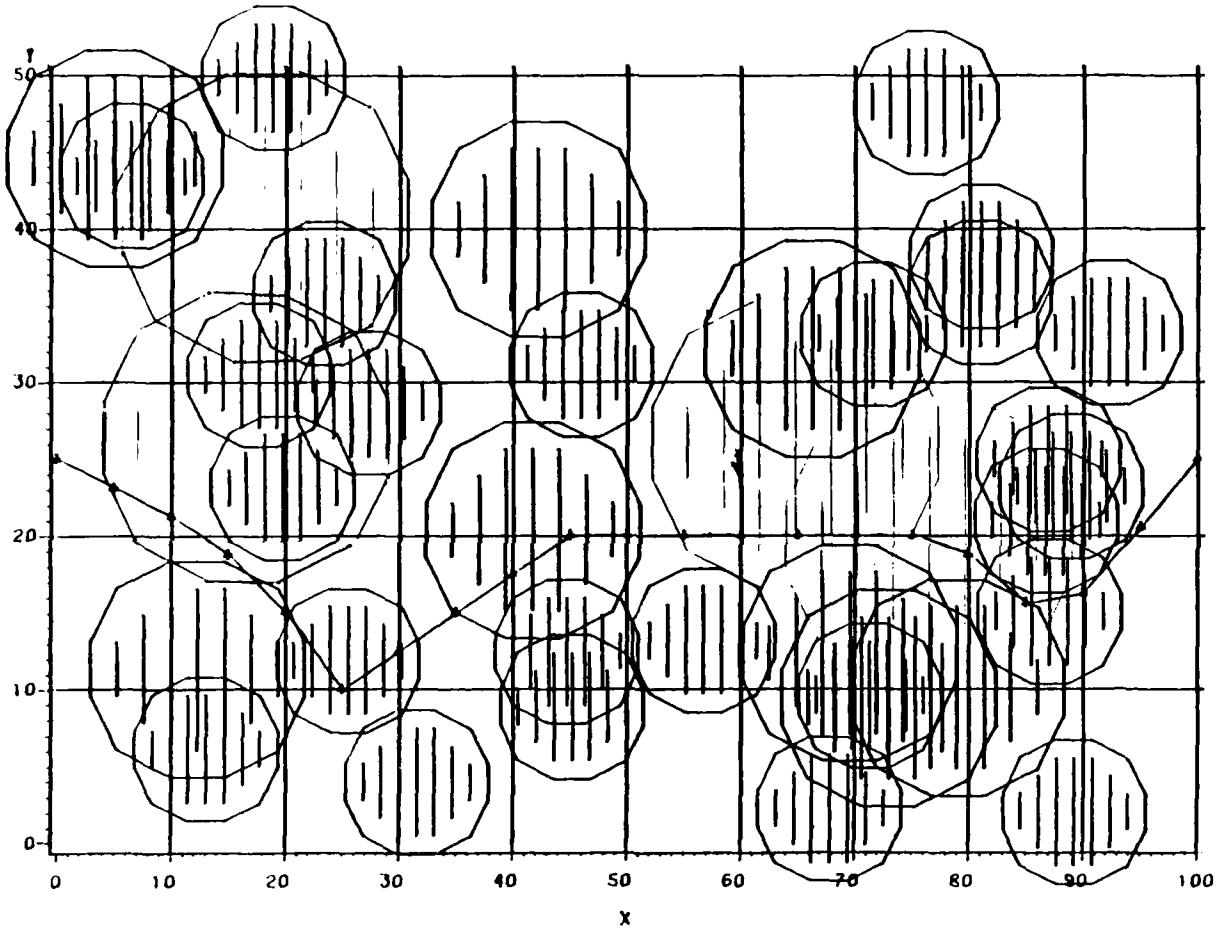


Figure 7.10

Path from Test 5 - 20 Divisions

Path Cost = 155

Path Length = 111.33

APPENDIX A

```
(* program hplan generates aircraft routes and finds a path through *)
(* random hostile environments using the artificial intelligence    *)
(* technique of hierarchical planning                               *)

program hplan(input, output, datain, pathout);

    const
                x_limit = 100;
        max_xboxes = 20;
        max_yboxes = 10;
        max_num_threats = 100;
        max_threats_per_box = 10;
        max_pts_on_line = 20;

    type
        block_size = (BIG, SMALL);
        cost_option = (ALL, DECONFLICT);
        coverage = (CENTER, TOTAL);
        threat_array = array[1..max_num_threats] of real;
        cost_array = array[1..max_num_threats] of integer;
        box_array = array[1..max_threats_per_box] of integer;
        grid_array = array[1..max_xboxes, 1..max_yboxes] of integer;
        threat_add_array = array[1..max_num_threats] of integer;
        path_array = array[1..max_xboxes] of integer;
        flight_path_array = array[1..max_pts_on_line, 1..max_xboxes] of real;
        final_flight_array = array[1..max_xboxes] of real;

        threat_values = record
                cx : threat_array;
                cy : threat_array;
                value : cost_array;
                radius : threat_array;
```

would be to select more than two block paths and then apply heuristics to search through the block paths for the best path. In any case, use of one or more artificial intelligence techniques for problem-solving is recommended.

miss threats altogether, while 20 divisions in that same block path may force the flight path to intersect a threat as was the case in test 5. The determining factor between using 10 or 20 divisions is the location of threats contained in a block path.

HPLAN implementation has some shortcomings. The quality of the final solution depends on block paths. An exhaustive search through a block path will find the best flight path within that block path. However, there is no guarantee that the block path chosen will contain the best flight path for the problem. With some threat environments, there may be more than two best block paths. Since only two are selected for exhaustive search, the best possible path may be contained in a block path not selected. Therefore, that portion of HPLAN that searches through a block path is admissable (a search algorithm that is guaranteed to find an optimal path to a goal, if one exists). However, the HPLAN algorithm as a whole is not admissable. The solution may be to use exhaustive search for all the best block paths. In some cases, this will be almost as expensive as an exhaustive search of the entire problem space. Another problem with HPLAN exists with the type path SMALL option. As discussed previously, the CPU time involved is undesirable. If CPU time is not a consideration, then the SMALL option may produce better paths.

Based on the results in this thesis, the hierarchical planning method was an excellent technique to use for solving this type of problem. The use of heuristics in [5] also has merit. Perhaps a combination of both methods would produce still better results. One possibility would be to use an 'intelligent' heuristic to choose the best block path and then perform an exhaustive search. Another approach

CONCLUSIONS

The result of using hierarchical planning to generate flight paths through a random threat environment produced better paths than did heuristic search in five of the eight test cases. If the results using the SMALL option with undesirable CPU time is included, the better paths were produced in six cases. This reinforces statements made earlier that planning increases problem-solving power. Several conclusions can be drawn from the test results. First, when using big block paths to solve the problem, it is better to evaluate the threat environment based on how the threats effect big boxes (x, y values at 10, 5). This method produced better results in six out of the eight tests. It would seem that evaluating the threats based on small boxes would provide a better representation of the threat environment. However, adding the small boxes together to process a big block path in some cases distorts actual threat concentration. The cost options ALL/DECONFLICT were different in three test cases. In two of the three cases, the DECONFLICT option produced a better path in terms of cost. This is because the DECONFLICT option only counts a threat cost once in determining the best block path. Intuitively, the DECONFLICT method of cost computation makes more sense. The box coverage options CENTER and TOTAL were equally effective. The number of divisions used in all the options sometimes made significant differences in path cost. As noted in test 3, 20 divisions will allow a path to maneuver around and between threats. However, 10 divisions will sometimes produce a flight path that will

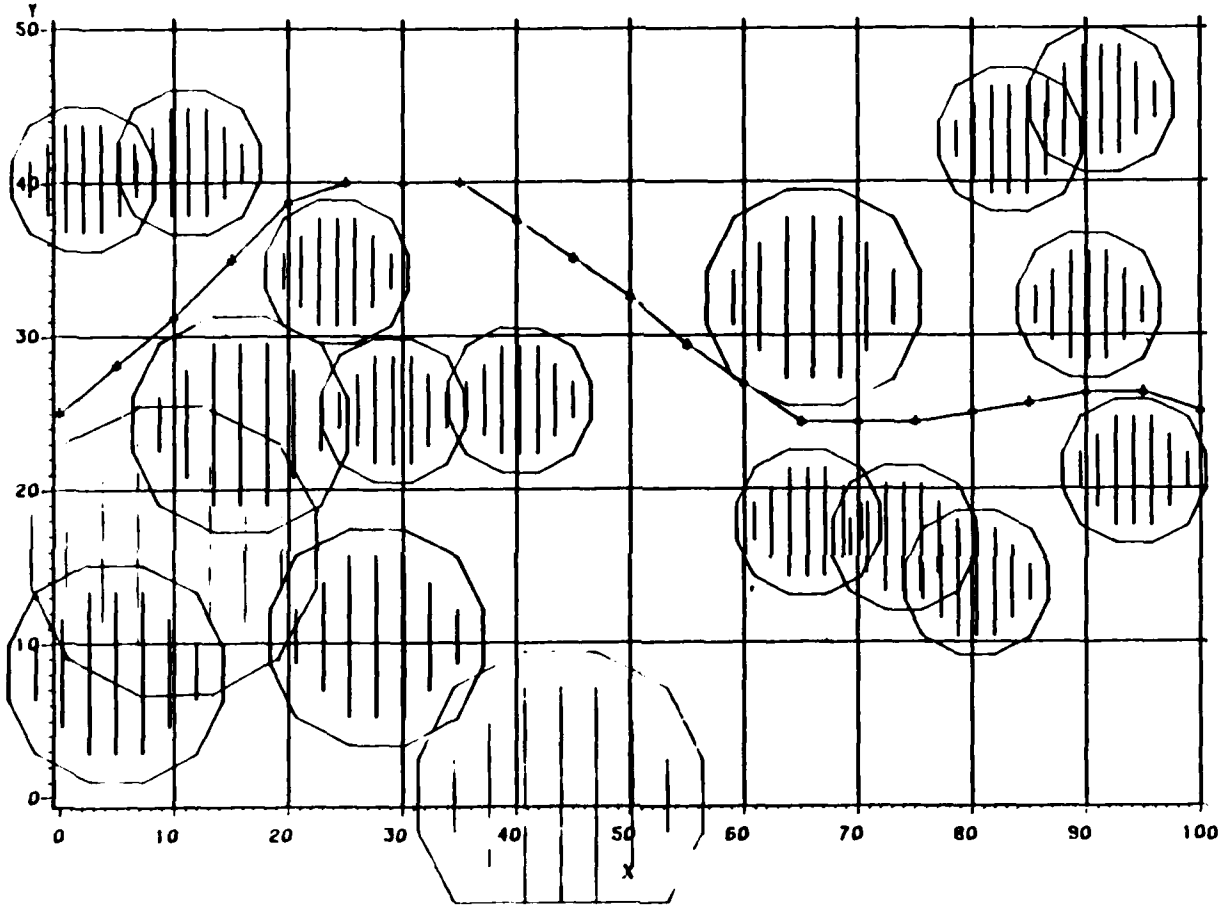


Figure 7.14

Path from Test 8 - 20 Divisions

Path Cost = 0

Path Length = 108.56

THREAT ENVIRONMENT:

RANDOM SEED: 39306
TOTAL THREAT DENSITY: 0.5
NUMBER OF THREATS: 18
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	9	50	108.86
BIG	ALL	TOTAL	10	5	10	9	18	50	108.54
BIG	ALL	TOTAL	10	5	10	17	53	50	107.74
BIG	ALL	TOTAL	10	5	20	5	14	0	111.83
BIG	ALL	TOTAL	10	5	20	9	35	0	109.49
BIG	ALL	TOTAL	10	5	20	17	114	0	108.56
BIG	DECON	TOTAL	10	5	10	5	11	50	108.86
BIG	DECON	TOTAL	10	5	10	9	21	50	108.54
BIG	DECON	TOTAL	10	5	10	17	55	50	107.74
BIG	DECON	TOTAL	10	5	20	5	17	0	111.83
BIG	DECON	TOTAL	10	5	20	9	38	0	109.49
BIG	DECON	TOTAL	10	5	20	17	116	0	108.56
BIG	ALL	TOTAL	20	10	10	5	9	50	108.86
BIG	ALL	TOTAL	20	10	10	9	19	50	108.54
BIG	ALL	TOTAL	20	10	10	17	54	50	107.91
BIG	ALL	TOTAL	20	10	20	5	14	0	111.83
BIG	ALL	TOTAL	20	10	20	9	35	0	109.49
BIG	ALL	TOTAL	20	10	20	17	115	0	108.94
BIG	ALL	CENTER	20	10	10	5	9	50	108.86
BIG	ALL	CENTER	20	10	10	9	18	50	108.54
BIG	ALL	CENTER	20	10	10	17	53	50	107.91
BIG	ALL	CENTER	20	10	20	5	14	0	111.83
BIG	ALL	CENTER	20	10	20	9	37	0	109.49
BIG	ALL	CENTER	20	10	20	17	113	0	108.94
HEURISTIC SEARCH RESULTS							54	0	109.49

Table 7.8

Results for Test 8

TEST 8

The paths from this test either avoid all the threats or intersect one threat with a cost of 50. In this test, the ALL/DECONFLICT cost options have the same results. In both cases, 20 divisions produce the better results. The TOTAL/CENTER box coverage options produce the same results and again 20 divisions is better than 10. This is another example where the number of divisions is significant. In all options, 20 divisions resulted in a path cost of 0, whereas the cost was 50 when the number of divisions is 10. This is because with 10 divisions, the path is unable to avoid a threat near the goal. When the number of divisions is 20, the path is able to set itself up for a move between the two threats. With 20 divisions, the flight legs are shorter and therefore the path is able to maneuver more. The best path for this test is illustrated in Figure 7.14. Heuristic search was also able to avoid all the threats, however, the path length was slightly longer.

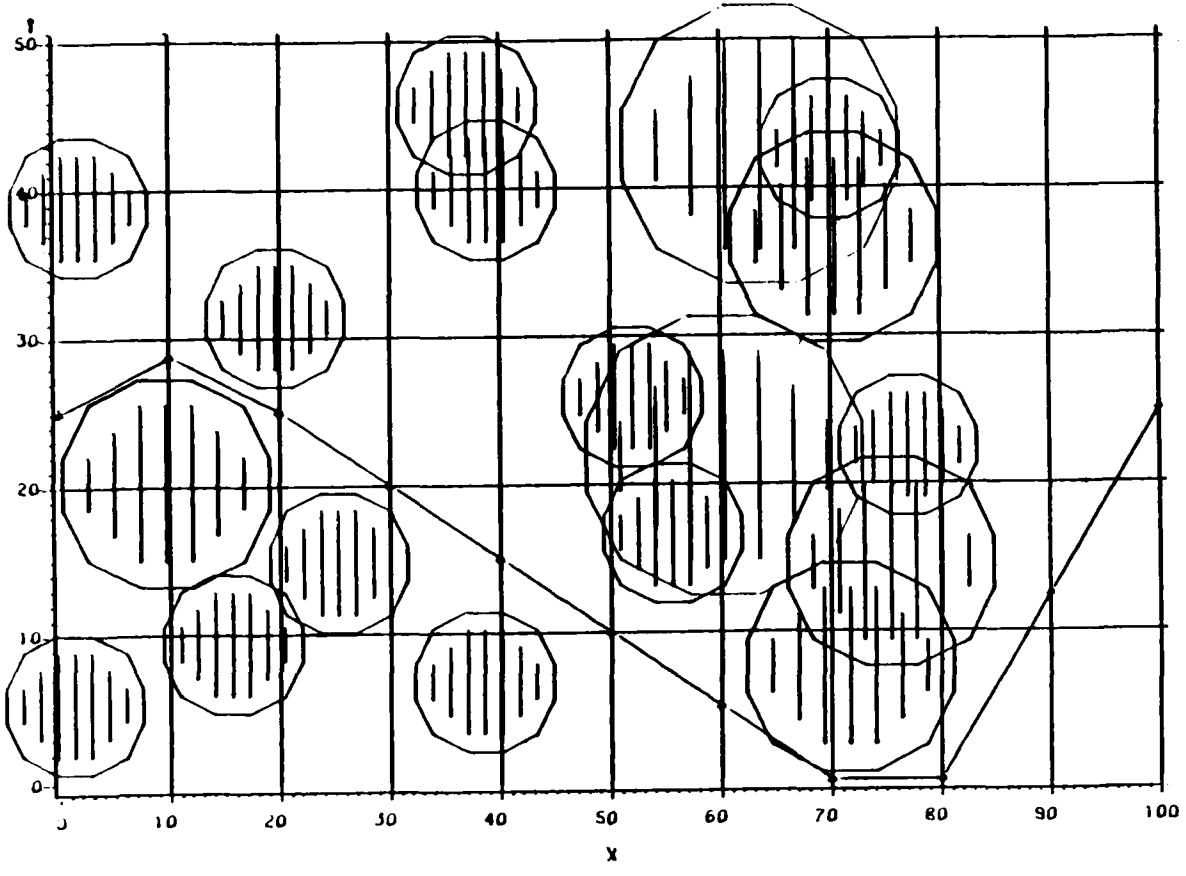


Figure 7.13

Path from Test 7 - DECONFLICT Option

Path Cost = 0

Path Length = 119.28

THREAT ENVIRONMENT:

RANDOM SEED: 28547
TOTAL THREAT DENSITY: 0.5
NUMBER OF THREATS: 18
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	9	10	118.94
BIG	ALL	TOTAL	10	5	10	9	18	10	118.62
BIG	ALL	TOTAL	10	5	10	17	54	10	117.55
BIG	ALL	TOTAL	10	5	20	5	14	50	117.40
BIG	ALL	TOTAL	10	5	20	9	36	50	111.57
BIG	ALL	TOTAL	10	5	20	17	117	50	110.07
BIG	DECON	TOTAL	10	5	10	5	11	0	120.43
BIG	DECON	TOTAL	10	5	10	9	20	0	119.28
BIG	DECON	TOTAL	10	5	10	17	55	0	119.28
BIG	DECON	TOTAL	10	5	20	5	16	0	132.07
BIG	DECON	TOTAL	10	5	20	9	37	0	126.22
BIG	DECON	TOTAL	10	5	20	17	116	0	124.08
BIG	ALL	TOTAL	20	10	10	5	9	10	116.29
BIG	ALL	TOTAL	20	10	10	9	18	10	114.01
BIG	ALL	TOTAL	20	10	10	17	54	10	112.87
BIG	ALL	TOTAL	20	10	20	5	14	10	123.10
BIG	ALL	TOTAL	20	10	20	9	37	10	120.75
BIG	ALL	TOTAL	20	10	20	17	117	10	119.45
BIG	ALL	CENTER	20	10	10	5	9	10	111.84
BIG	ALL	CENTER	20	10	10	9	19	10	105.07
BIG	ALL	CENTER	20	10	10	17	54	10	104.76
BIG	ALL	CENTER	20	10	20	5	14	10	114.19
BIG	ALL	CENTER	20	10	20	9	36	10	108.14
BIG	ALL	CENTER	20	10	20	17	121	10	106.20
HEURISTIC SEARCH RESULTS							3	10	104.57

Table 7.7

Results for Test 7

TEST 7

There were six different block paths produced by the options in this test case. Block paths in this test covered virtually all possible ways of avoiding the threats. This is due to the small threat density that produced only 18 threats. Therefore, there are many ways for block paths to maneuver around threats. Some went to the top of the grid, while others stayed in the middle or at the bottom. The best block path was with x, y at 10, 5 and the cost option equal to DECONFLICT. This block path produced a flight path that avoided all threats in the grid to give a cost of 0. The best the heuristic search could produce was a path with a cost of 10. Again, the results for the cost options ALL and DECONFLICT were different, with DECONFLICT having better results. The option with x, y at 20, 10 also produced different results with the CENTER/TOTAL box coverage options. This time the CENTER option produced a shorter flight path. The best flight path for this test case is given in Figure 7.13. This path runs on the x-axis for 10 units, thereby just missing a threat. The y coordinate for the threat center just referred to is 7.61. Since the radius of this threat is 7.5, that leaves just 0.11 units for the flight path to avoid the threat. The path in Figure 7.13 accomplishes this.

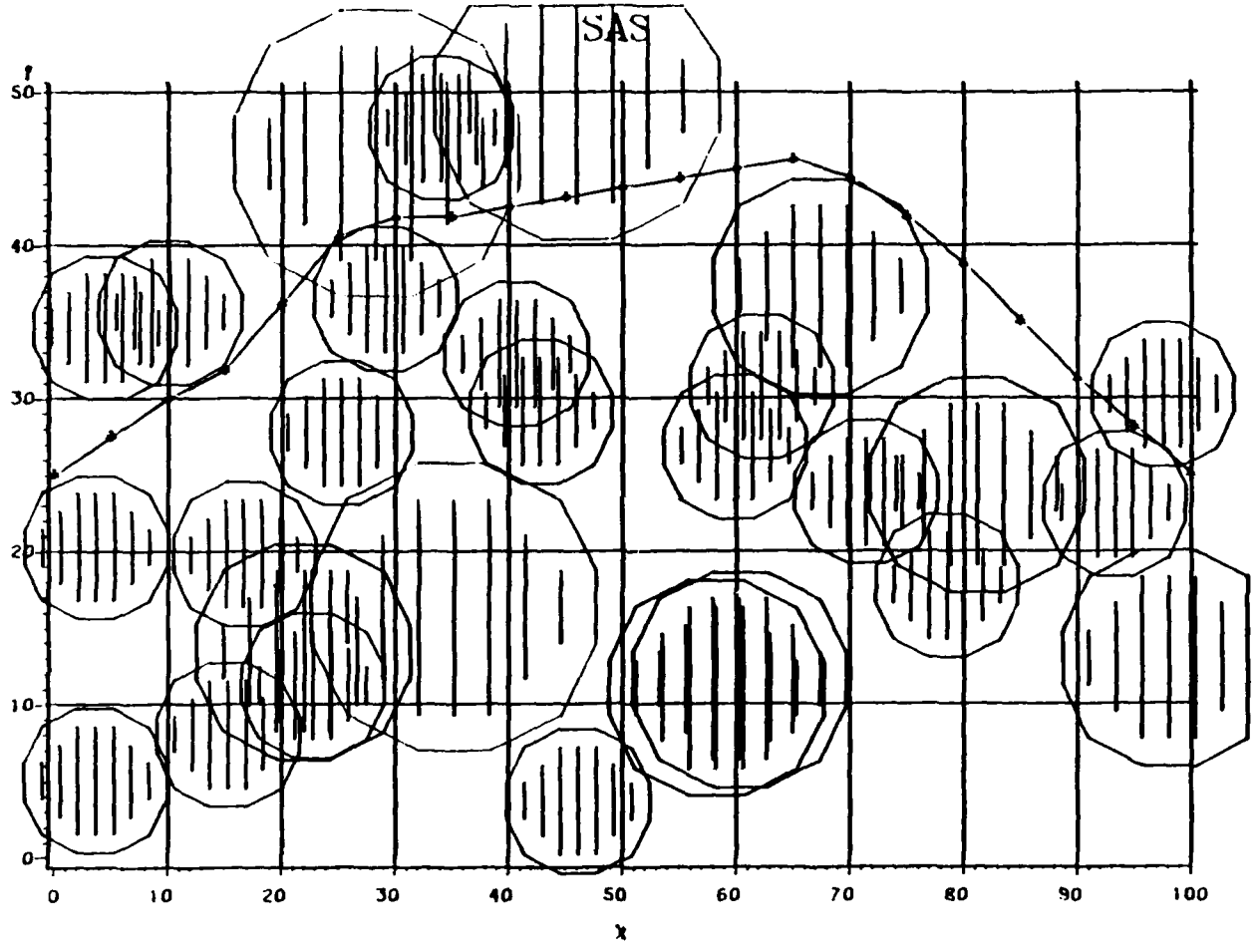


Figure 7.12

Path from Test 6 - 20 Divisions

Path Cost = 70

Path Length = 111.13

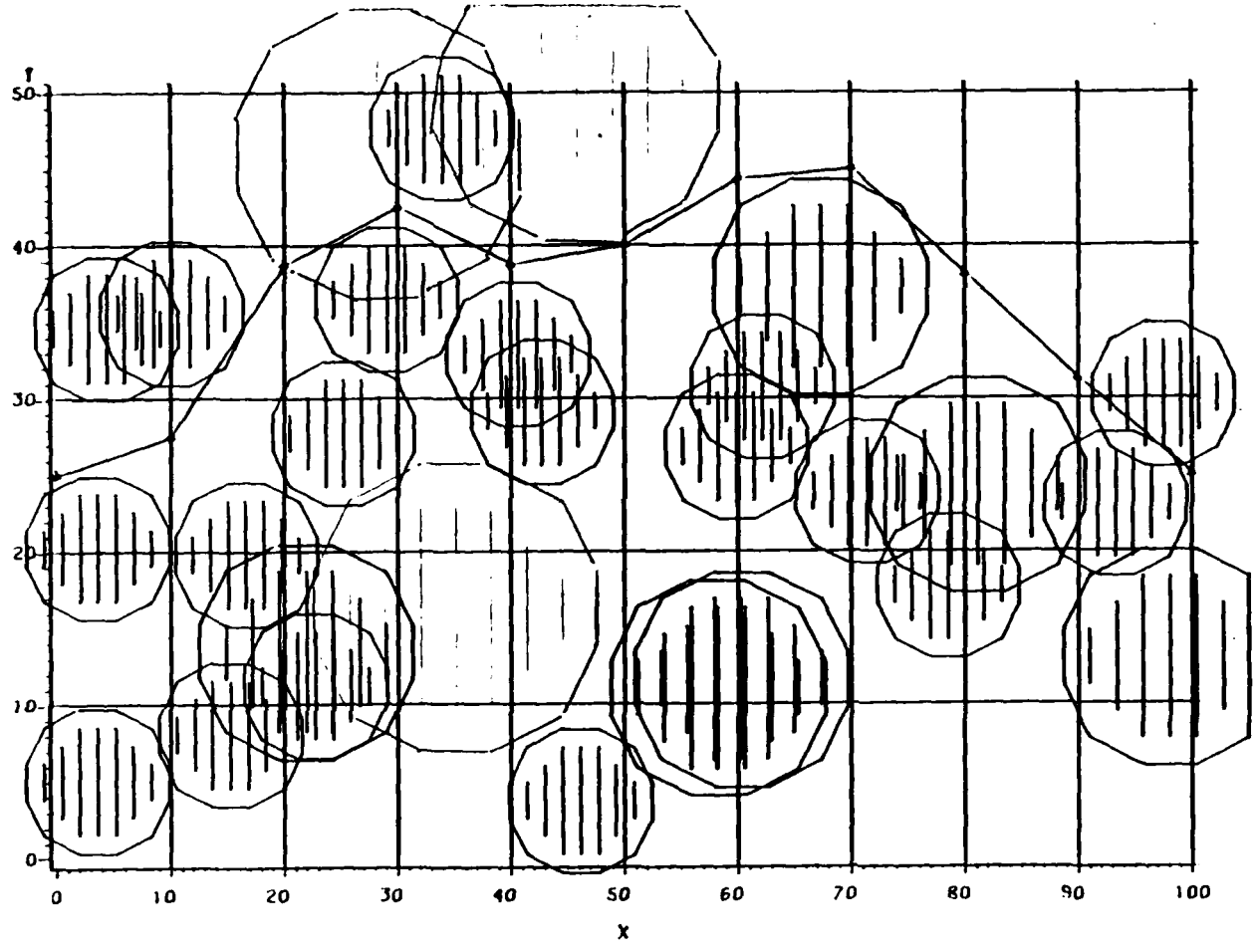


Figure 7.11

Path from Test 6 - TOTAL Option

Path Cost = 60

Path Length = 113.80

THREAT ENVIRONMENT:

RANDOM SEED: 120775
TOTAL THREAT DENSITY: 0.75
NUMBER OF THREATS: 28
THREAT TYPES: 3

RADIUS	COST	DENSITY
5	50	0.4
7.5	25	0.3
10	10	0.3

RESULTS:

TYPE PATH	COST OPTION	BOX COVERAGE	X	Y	DIV	PTS/DIV	CPU	COST	LENGTH
BIG	ALL	TOTAL	10	5	10	5	11	110	111.88
BIG	ALL	TOTAL	10	5	10	9	26	70	125.65
BIG	ALL	TOTAL	10	5	10	17	79	70	122.89
BIG	ALL	TOTAL	10	5	20	5	19	95	121.26
BIG	ALL	TOTAL	10	5	20	9	54	60	117.63
BIG	ALL	TOTAL	10	5	20	17	173	60	116.62
BIG	DECON	TOTAL	10	5	10	5	14	110	111.88
BIG	DECON	TOTAL	10	5	10	9	30	70	125.65
BIG	DECON	TOTAL	10	5	10	17	83	70	122.89
BIG	DECON	TOTAL	10	5	20	5	22	95	121.26
BIG	DECON	TOTAL	10	5	20	9	55	60	117.63
BIG	DECON	TOTAL	10	5	20	17	174	60	116.62
BIG	ALL	TOTAL	20	10	10	5	11	110	112.44
BIG	ALL	TOTAL	20	10	10	9	25	110	109.68
BIG	ALL	TOTAL	20	10	10	17	78	60	113.80
BIG	ALL	TOTAL	20	10	20	5	20	120	113.01
BIG	ALL	TOTAL	20	10	20	9	51	70	113.13
BIG	ALL	TOTAL	20	10	20	17	171	70	112.25
BIG	ALL	CENTER	20	10	10	5	11	110	112.44
BIG	ALL	CENTER	20	10	10	9	26	70	113.72
BIG	ALL	CENTER	20	10	10	17	79	60	113.82
BIG	ALL	CENTER	20	10	20	5	19	70	114.36
BIG	ALL	CENTER	20	10	20	9	52	70	112.23
BIG	ALL	CENTER	20	10	20	17	172	70	111.13
HEURISTIC SEARCH RESULTS							810	60	106.82

Table 7.6

Results for Test 6

TEST 6

This was the first test case where the box coverage options CENTER and TOTAL produced different results. With x, y values equal to 20, 10, the TOTAL option resulted in the best path. This is shown in Figure 7.11. In both CENTER and TOTAL options, the best path resulted when the number of divisions was 10. When the number of divisions was 20, the flight paths intersected an additional threat with a cost of 10. The best path with 20 divisions with x, y at 20, 10 is shown in Figure 7.12. The heuristic search method also found a path with a cost of 60, but it was shorter than any of the above paths.

```
        end;

next_point = record
        xcoor : flight_path_array;
        ycoor : flight_path_array;
        end;

flight_paths = record
        xpt : flight_path_array;
        ypt : flight_path_array;
        conflict : flight_path_array;
        len : flight_path_array;
        end;

threat_rec = record
        num : integer;
        value : box_array;
        kind : box_array;
        end;

box_threat = array[1..max_xboxes, 1..max_yboxes] of threat_rec;

threat_add = record
        add : threat_add_array;
        x : threat_add_array;
        y : threat_add_array;
        end;

block_rec = record
        x : path_array;
        y : path_array;
        score : integer;
        end;

block_path = array[1..max_xboxes] of block_rec;
```

```
        flight_rec = record
                x : final_flight_ar ay;
                y : final_flight_array;
                cost : real;
                len : real;
                end;

        final_flight = array[1..max_xboxes] of flight_rec;

    var
        threat_in : threat_values;
        threat, visit : grid_array;
        cur_xpath, cur_ypath, best_xpath, best_ypath,
          next_best_xpath, next_best_ypath : path_array;
        next_pt : next_point;
        path : flight_paths;
        threat_decon : box_threat;
        threat_added : threat_add;
        datain, pathout : text;
        type_path : block_size;
        block : block_path;
        flight : final_flight;
        option : cost_option;
        box_coverage : coverage;
        x_max, y_max, tot_cpu_time, threat_num,
          num_div, pts_on_line, path_num : integer;
        first_xpt, first_ypt, last_xpt, last_ypt : real;

(* distance computes the distance between two points *)

function distance(x1, y1, x2, y2 : real) : real;

    begin
    distance := sqrt((x1 - x2) ** 2 + (y1 - y2) ** 2);
    end;
```

```
(* initialize intializes data structures for processing *)

procedure initialize;

    var
            i, j, k : integer;
    begin
    reset(datain);
    rewrite(pathout);
    threat_num := 0;
    path_num := 0;
    first_xpt := 0.0;
    first_ypt := 25.0;
    last_xpt := 100.0;
    last_ypt := 25.0;
    for i:= 1 to max_xboxes do
            for j := 1 to max_yboxes do
                    threat[i, j] := 0;
    for i := 1 to max_xboxes do
            begin
            for j := 1 to max_yboxes do
                    begin
                    threat_decon[i, j].num := 0;
                    for k := 1 to max_threats_per_box do
                            begin
                            threat_decon[i, j].value[k] := 0;
                            threat_decon[i, j].kind[k] := 0;
                            end;
                    end;
            end;
    for i := 1 to max_num_threats do
            begin
            threat_added.add[i] := 0;
            threat_added.x[i] := 0;
            threat_added.y[i] := 0;
            end;
```

```
    end;

(* print_inputs prints the program options selected *)
(* and the threat environment                       *)

procedure print_inputs;
    var
            i : integer;
    begin
    writeln(pathout, 'HPLAN RESULTS:');
    writeln(pathout);
    writeln(pathout);
    writeln(pathout, 'OPTIONS SELECTED:');
    writeln(pathout);
    writeln(pathout, '   TYPE OF BLOCK PATHS -- ', type_path);
    writeln(pathout, '   BLOCK PATH COSTS    -- ', option);
    writeln(pathout, '   BOX THREAT COVERAGE -- ', box_coverage);
    writeln(pathout, '   XY BLOCK LIMITS     -- ', x_max : 3, y_max : 3);
    writeln(pathout, '   NUMBER OF DIVISIONS -- ', num_div : 3);
    writeln(pathout, '   POINTS PER DIVISION -- ', pts_on_line : 3);
    writeln(pathout);
    writeln(pathout);
    writeln(pathout, ' INPUT THREAT ENVIRONMENT:');
    writeln(pathout);
    writeln(pathout, '   TOTAL NUMBER OF THREATS -- ', threat_num : 1);
    writeln(pathout);
    writeln(pathout, '   THREAT INPUTS -- X Y RADIUS COST');
    writeln(pathout);
    for i := 1 to threat_num do
            writeln(pathout, '      ', threat_in.cx[i] : 8:2,
                    threat_in.cy[i] : 8:2, threat_in.radius[i] : 6:1,
                    threat_in.value[i] : 5);
    page(pathout);
    end;

(* read_data reads input data from file datain built by threat_bldr *)
```

```
procedure read_data;

    var
            x_in, y_in, t_rad : real;
            t_val : integer;
    begin
    readln(datain, type_path);
    readln(datain, option);
    readln(datain, box_coverage);
    readln(datain, x_max, y_max);
    readln(datain, num_div, pts_on_line);
    readln(datain, x_in, y_in, t_val, t_rad);
    while (x_in <> -1) do
            (* read and store all threats *)
            begin
            threat_num := threat_num + 1;
            threat_in.cx[threat_num] := x_in;
            threat_in.cy[threat_num] := y_in;
            threat_in.value[threat_num] := t_val;
            threat_in.radius[threat_num] := t_rad;
            readln(datain, x_in, y_in, t_val, t_rad);
            end;
    print_inputs;
    end;

(* compute_block_coor takes the center of a threat and returns the *)
(* xy box coordinate where the center of the threat is located     *)

procedure compute_block_coor(x_in, y_in : real; var x, y : integer);

    var
            x_rnd, y_rnd, remand, units : integer;
    begin
    units := x_limit div x_max;
    x_rnd := round(x_in);
```

```
    y_rnd := round(y_in);
    if x_rnd < 1 then
            x_rnd := 1;
    if y_rnd < 1 then
            y_rnd := 1;
    remand := x_rnd mod units;
    x := x_rnd div units;
    if remand <> 0 then
            x := x + 1;
    remand := y_rnd mod units;
    y := y_rnd div units;
    if remand <> 0 then
            y := y + 1;
    end;

(* check_box calculates the distance between the center of a box *)
(* and the center of a threat, and if the distance is less than  *)
(* or equal to the threat radius, the threat value is assigned   *)
(* to the box                                                    *)

procedure check_box(x, y, inx : integer);

    var
            i, units : integer;
            half_units, x_mid, y_mid, dist : real;
    begin
    units := x_limit div x_max;
    half_units := units / 2;
    x_mid := x * units - half_units;  (* calculate the xy coordinates *)
    y_mid := y * units - half_units;  (* for the center of a box      *)
    dist := distance(threat_in.cx[inx], threat_in.cy[inx], x_mid, y_mid);
    if dist <= threat_in.radius[inx] then
            (* threat is contained in the box *)
            begin
            if option = DECONFLICT then
                    begin
```

```
                    i := threat_decon[x, y].num + 1;
                    threat_decon[x, y].num := i;
                    threat_decon[x, y].value[i] := threat_in.value[inx];
                    threat_decon[x, y].kind[i] := inx;
                    end
            else
                    threat[x, y] := threat[x, y] + threat_in.value[inx];
            end;
    end;

(* threat_analysis takes the xy box coordinate containing a threat *)
(* and checks adjacent boxes for containment in the threat radius  *)

procedure threat_analysis(x, y, threat_inx : integer);

    var
            small_boxes : boolean;
    begin
    if (x_max = 20) and (box_coverage = TOTAL) then
            small_boxes := true
    else
            small_boxes := false;
    (* check boxes above *)
    if (y + 1) <= y_max then
            begin
            check_box(x, y + 1, threat_inx);
            if (small_boxes) and (y + 2 <= y_max) then
                    check_box(x, y + 2, threat_inx);
            end;
    (* check boxes below *)
    if (y - 1) >= 1 then
            begin
            check_box(x, y - 1, threat_inx);
            if (small_boxes) and (y - 2 >= 1) then
                    check_box(x, y - 2, threat_inx);
            end;
```

```
(* check boxes to the left *)
if (x - 1) >= 1 then
        begin
        check_box(x - 1, y, threat_inx);
        if (small_boxes) and (x - 2 >= 1) then
                check_box(x - 2, y, threat_inx);
        end;
(* check boxes to the right *)
if (x + 1) <= x_max then
        begin
        check_box(x + 1, y, threat_inx);
        if (small_boxes) and (x + 2 <= x_max) then
                check_box(x + 2, y, threat_inx);
        end;
(* check boxes in the upper left *)
if ((x - 1 >= 1) and (y + 1 <= y_max)) then
        begin
        check_box(x - 1, y + 1, threat_inx);
        if (small_boxes) then
                begin
                if (x - 2 >= 1) and (y + 1 <= y_max) then
                        check_box(x - 2, y + 1, threat_inx);
                if (x - 1 >= 1) and (y + 2 <= y_max) then
                        check_box(x - 1, y + 2, threat_inx);
                if (x - 2 >= 1) and (y + 2 <= y_max) then
                        check_box(x - 2, y + 2, threat_inx);
                end;
        end;
(* check boxes in the upper right *)
if ((x + 1 <= x_max) and (y + 1 <= y_max)) then
        begin
        check_box(x + 1, y + 1, threat_inx);
        if (small_boxes) then
                begin
                if (x + 2 <= x_max) and (y + 1 <= y_max) then
                        check_box(x + 2, y + 1, threat_inx);
```

```
                if (x + 1 <= x_max) and (y + 2 <= y_max) then
                        check_box(x + 1, y + 2, threat_inx);
                if (x + 2 <= x_max) and (y + 2 <= y_max) then
                        check_box(x + 2, y + 2, threat_inx);
                end;
        end;
(* check boxes in the lower left *)
if ((x - 1 >= 1) and (y - 1 >= 1)) then
        begin
        check_box(x - 1, y - 1, threat_inx);
        if (small_boxes) then
                begin
                if (x - 2 >= 1) and (y - 1 >= 1) then
                        check_box(x - 2, y - 1, threat_inx);
                if (x - 1 >= 1) and (y - 2 >= 1) then
                        check_box(x - 1, y - 2, threat_inx);
                if (x - 2 >= 1) and (y - 2 >= 1) then
                        check_box(x - 2, y - 2, threat_inx);
                end;
        end;
(* check boxes in the lower right *)
if ((x + 1 <= x_max) and (y - 1 >= 1)) then
        begin
        check_box(x + 1, y - 1, threat_inx);
        if (small_boxes) then
                begin
                if (x + 2 <= x_max) and (y - 1 >= 1) then
                        check_box(x + 2, y - 1, threat_inx);
                if (x + 1 <= x_max) and (y - 2 >= 1) then
                        check_box(x + 1, y - 2, threat_inx);
                if (x + 2 <= x_max) and (y - 2 >= 1) then
                        check_box(x + 2, y - 2, threat_inx);
                end;
        end;
end;
```

```
(* print_threats prints a matrix of the grid with the threat values *)
(* that are assigned to the boxes                                   *)

procedure print_threats;

    var
            x, y, i : integer;
    begin
    writeln(pathout);
    writeln(pathout, 'BOX THREAT VALUES');
    writeln(pathout);
    if option = DECONFLICT then
            begin
            for y := y_max downto 1 do
                    begin
                    write(pathout, y : 3);
                    for i := 1 to max_threats_per_box do
                            begin
                            for x := 1 to x_max do
                                    write(pathout, threat_decon[x,y].value[i] : 3, '   ');
                            writeln(pathout);
                            if i <> max_threats_per_box then
                                    write(pathout, '   ');
                            end;
                    end;
            end
    else
            begin
            for y := y_max downto 1 do
                    begin
                    write(pathout, y : 3);
                    for x := 1 to x_max do
                            write(pathout, threat[x, y] : 3, '   ');
                    writeln(pathout);
                    end;
            end;
```

```
    for x := 1 to x_max do
            write(pathout, '    ', x : 3);
    writeln(pathout);
    end;

(* add_small_boxes adds the values in small boxes to obtain values for *)
(* big boxes.  Each big box contains four small boxes                   *)

procedure add_small_boxes;

    var
            i, j, x, y : integer;
            sm_threat : grid_array;
    begin
    (* store values in array sm_threat *)  -
    for i := 1 to x_max do
            for j := 1 to y_max do
                    sm_threat[i, j] := threat[i, j];
    j := 0;
    (* reset grid from small to big boxes *)
    x_max := 10;
    y_max := 5;
    for x := 1 to x_max do
            begin
            i := 0;
            for y := 1 to y_max do
                    begin
                    (* add four small boxes to get big box values *)
                    threat[x, y] := sm_threat[x + j, y + i] +
                                    sm_threat[x + j, y + i + 1] +
                                    sm_threat[x + j + 1, y + i] +
                                    sm_threat[x + j + 1, y + i + 1];
                    i := i + 1;
                    end;
            j := j + 1;
            end;
```

```
    end;

(* process_threats evaluates the threat environment and places *)
(* threat values in the boxes                                  *)

procedure process_threats;

    var
            i, j, x, y : integer;
    begin
    tot_cpu_time := clock;  (* system clock *)
    for i := 1 to threat_num do
            (* process each threat *)
            begin
            compute_block_coor(threat_in.cx[i], threat_in.cy[i], x, y);
            if option = DECONFLICT then
                    (* store threat value in box *)
                    begin
                    j := threat_decon[x, y].num + 1;
                    threat_decon[x, y].num := j;
                    threat_decon[x, y].value[j] := threat_in.value[i];
                    threat_decon[x, y].kind[j] := i
                    end
            else (* add in threat to box value *)
                    threat[x, y] := threat[x, y] + threat_in.value[i];
            threat_analysis(x, y, i);
            end;
    print_threats;
    if (type_path = BIG) and (x_max = 20) then
            (* add up small box values to get big box values *)
            begin
            add_small_boxes;
            print_threats;
            end;
    end;
```

```
(* block_paths computes and stores the best block paths *)

procedure block_paths;

    const
            max_ties = 10;

    type
            ties = record
                    x : path_array;
                    y : path_array;
                    end;

            tie_array = array[1..max_ties] of ties;

    var
            x, y, inx, cur_score, best_score,
              next_best_score, num_ties : integer;
            tie_paths : tie_array;

(* add_to_list adds a box to the current block path *)

procedure add_to_list(x, y : integer);

    begin
    inx := inx + 1;
    cur_xpath[inx] := x;
    cur_ypath[inx] := y;
    end;

(* init_block_paths initializes the structures needed *)
(* for block path computation                         *)

procedure init_block_paths;

    var
```

```
        i, j : integer;
begin
x := 1;
if type_path = SMALL then
        y := 5
else
        y := 3;
inx := 0;
num_ties := 0;
add_to_list(x, y);
(* initialize beginning score for block paths *)
if option = DECONFLICT then
        begin
        cur_score := 0;
        if threat_decon[x, y].num <> 0 then
                begin
                for i := 1 to threat_decon[x, y].num do
                        begin
                        cur_score := cur_score + threat_decon[x, y].value[i];
                        j := threat_decon[x, y].kind[i];
                        threat_added.add[j] := 1;
                        threat_added.x[j] := x;
                        threat_added.y[j] := y;
                        end;
                end;
        end
else
        cur_score := threat[x, y];
best_score := 100000;
next_best_score := 100001;
(* initialize all boxes to not visited yet *)
for i := 1 to x_max do
        for j := 1 to y_max do
                visit[i, j] := 0;
visit[x, y] := 1;
(* initialize array to hold the best block path *)
```

```
    flight[path_num].cost := path.conflict[inx, num_div];
    flight[path_num].len := path.len[inx, num_div];
    end;

(* compute_best_flight_path locates the best flight path generated *)

procedure compute_best_flight_path(var inx : integer);

    var
            score, dist : real;
            i : integer;
    begin
    score := path.conflict[1, num_div];
    dist := path.len[1, num_div];
    inx := 1;
    for i := 2 to pts_on_line do
            (* process each path *)
            begin
            if path.conflict[i, num_div] <= score then
                    (* best path so far *)
                    begin
                    if path.conflict[i, num_div] = score then
                            begin
                            if path.len[i, num_div] < dist then
                                    begin
                                    inx := i;
                                    dist := path.len[i, num_div];
                                    score := path.conflict[i, num_div];
                                    end;
                            end
                    else
                            begin
                            inx := i;
                            dist := path.len[i, num_div];
                            score := path.conflict[i, num_div];
                            end;
```

```
            next_best_xpath[i] := tie_paths[inx2].x[i];
            next_best_ypath[i] := tie_paths[inx2].y[i];
            end;
    next_best_score := best_score;
    end;


    begin  (* main section of block_paths *)
    init_block_paths;
    best_block_paths(x, y);
    if num_ties > 0 then
            resolve_ties;
    save_best_block_paths;
    end;

(* exhaust performs an exhaustive search of block paths *)
(* to find the best flight path                         *)

procedure exhaust;

    var
            inx : integer;
            temp_path : flight_paths;

(* save_flight saves a flight path, cost, and length *)

procedure save_flight(path_num, inx : integer);

    var
            i : integer;
    begin
    for i := 1 to num_div do
            begin
            flight[path_num].x[i] := path.xpt[inx, i];
            flight[path_num].y[i] := path.ypt[inx, i];
            end;
```

```
(* resolve_ties considers all block paths with the same best *)
(* scores and chooses the two that are the most different    *)

procedure resolve_ties;
    var
            i, j, diff, best_diff, inx1, inx2 : integer;
    begin
    (* put best block path with the ties *)
    num_ties := num_ties + 1;
    inx1 := 1;
    inx2 := 2;
    for i := 1 to x_max do
            begin
            tie_paths[num_ties].x[i] := best_xpath[i];
            tie_paths[num_ties].y[i] := best_ypath[i];
            end;
    best_diff := 0;
    (* find the two paths most different *)
    for i := 1 to num_ties - 1 do
            begin
            for j := i + 1 to num_ties do
                    begin
                    diff := compute_diff(i, j);
                    if diff > best_diff then
                            begin
                            best_diff := diff;
                            inx1 := i;
                            inx2 := j;
                            end;
                    end;
            end;
    (* store the two most different *)
    for i := 1 to x_max do
            begin
            best_xpath[i] := tie_paths[inx1].x[i];
            best_ypath[i] := tie_paths[inx1].y[i];
```

```
                        best_block_paths(x, y);
                end;
        end;
    end;

(* best_block_paths is the main driver that checks all block paths *)
(* in the grid to determine the best block paths                   *)

procedure best_block_paths;

    begin
    path_up(x, y);
    path_next(x, y);
    path_down(x, y);
    visit[x, y] := 0;
    if option = DECONFLICT then
            cur_score := score_minus(cur_score, x, y)
    else
            cur_score := cur_score - threat[x, y];
    inx := inx - 1;
    end;

(* compute_diff computes the difference between the y coordinates *)
(* of two block paths and returns the score                       *)

function compute_diff(inx1, inx2 : integer) : integer;
    var
            score, i : integer;
    begin
    score := 0;
    for i := 1 to x_max do
            score := score + abs(tie_paths[inx1].y[i] -
                                 tie_paths[inx2].y[i]);
    compute_diff := score;
    end;
```

```
 (threat_decon[x + 1, y - 1].value[1] <> -1) and
 (visit[x + 1, y - 1] = 0)) or
((option = ALL) and (threat[x + 1, y - 1] >= 0) and
 (visit[x + 1, y - 1] = 0)) then
     (* box available for processing and has *)
     (* not been visited on current path     *)
     begin
     x := x + 1;
     y := y - 1;
     if (option = DECONFLICT) then
             cur_score := score_plus(cur_score, x, y)
     else
             cur_score := cur_score + threat[x, y];
     add_to_list(x, y);
     visit[x, y] := 1;
     if ((x_max = 10) and (x = 10) and (y = 3)) or
        ((x_max = 20) and (x = 20) and (y = 5)) then
             (* have reached goal for current path *)
             begin
             if (cur_score = best_score) and (num_ties < max_ties - 1) then
                     begin
                     num_ties := num_ties + 1;
                     save_tie;
                     end;
             if cur_score < best_score then
                     new_best(1)
             else if cur_score < next_best_score then
                     new_best(2);
             visit[x, y] := 0;
             if option = DECONFLICT then
                     cur_score := score_minus(cur_score, x, y)
             else
                     cur_score := cur_score - threat[x, y];
             inx := inx - 1;
             end
     else (* continue processing current block path *)
```

```
                    add_to_list(x, y);
                    visit[x, y] := 1;
                    if ((x_max = 10) and (x = 10) and (y = 3)) or
                       ((x_max = 20) and (x = 20) and (y = 5)) then
                            (* have reached goal for current path *)
                            begin
                            if (cur_score = best_score) and (num_ties < max_ties - 1) then
                                    begin
                                    num_ties := num_ties + 1;
                                    save_tie;
                                    end;
                            if cur_score < best_score then
                                    new_best(1)
                            else if cur_score < next_best_score then
                                    new_best(2);
                            visit[x, y] := 0;
                            if option = DECONFLICT then
                                    cur_score := score_minus(cur_score, x, y)
                            else
                                    cur_score := cur_score - threat[x, y];
                            inx := inx - 1;
                            end
                    else (* continue processing current block path *)
                            best_block_paths(x, y);
                    end;
            end;
    end;

(* path_down processes the next box down in the current block path *)

procedure path_down(x, y : integer);

    begin
    if ((x + 1 <= x_max) and (y - 1 >= 1)) then
            begin
            if ((option = DECONFLICT) and
```

```
                                new_best(1)
                            else if cur_score < next_best_score then
                                new_best(2);
                            visit[x, y] := 0;
                            if option = DECONFLICT then
                                cur_score := score_minus(cur_score, x, y)
                            else
                                cur_score := cur_score - threat[x, y];
                            inx := inx - 1;
                            end
                        else (* continue processing current block path *)
                            best_block_paths(x, y);
                        end;
                end;
        end;

(* path_next processes the box next to the current box *)

procedure path_next(x, y : integer);

    begin
    if (x + 1 <= x_max) then
            begin
            if ((option = DECONFLICT) and
                (threat_decon[x + 1, y].value[1] <> -1) and
                (visit[x + 1, y] = 0)) or
               ((option = ALL) and (threat[x + 1, y] >= 0) and
                (visit[x + 1, y] = 0)) then
                    (* box available for processing and has *)
                    (* not been visited on current path     *)
                    begin
                    x := x + 1;
                    if (option = DECONFLICT) then
                            cur_score := score_plus(cur_score, x, y)
                    else
                            cur_score := cur_score + threat[x, y];
```

```
procedure best_block_paths(x, y : integer); forward;

(* path_up processes the next box up in the current block path *)

procedure path_up(x, y : integer);

    begin
    if ((x + 1 <= x_max) and (y + 1 <= y_max)) then
            begin
            if ((option = DECONFLICT) and
                (threat_decon[x + 1, y + 1].value[1] <> -1) and
                (visit[x + 1, y + 1] = 0)) or
               ((option = ALL) and (threat[x + 1, y + 1] >= 0) and
                (visit[x + 1, y + 1] = 0)) then
                    (* box available for processing and has *)
                    (* not been visited on current path     *)
                    begin
                    x := x + 1;
                    y := y + 1;
                    if (option = DECONFLICT) then
                            cur_score := score_plus(cur_score, x, y)
                    else
                            cur_score := cur_score + threat[x, y];
                    add_to_list(x, y);
                    visit[x, y] := 1;
                    if ((x_max = 10) and (x = 10) and (y = 3)) or
                       ((x_max = 20) and (x = 20) and (y = 5)) then
                            (* have reached goal for current path *)
                            begin
                            if (cur_score = best_score) and (num_ties < max_ties - 1) then
                                    begin
                                    num_ties := num_ties + 1;
                                    save_tie;
                                    end;
                            if cur_score < best_score then
```

```
    if threat_decon[x, y].num <> 0 then
            (* there are threat values in this box *)
            begin
            for i := 1 to threat_decon[x, y].num do
                    (* process each threat value for the box *)
                    begin
                    k := threat_decon[x, y].kind[i];
                    if (threat_added.add[k] = 1) and
                       (threat_added.x[k] = x) and
                       (threat_added.y[k] = y) then
                            (* subtract threat value from score *)
                            (* and mark as not added            *)
                            begin
                            score := score - threat_decon[x, y].value[i];
                            threat_added.add[k] := 0;
                            threat_added.x[k] := 0;
                            threat_added.y[k] := 0;
                            end;
                    end;
            end;
    score_minus := score;
    end;

(* save_tie saves the current block path that has the *)
(* same score as the best block path so far          *)

procedure save_tie;
    var
            i : integer;
    begin
    for i := 1 to x_max do
            begin
            tie_paths[num_ties].x[i] := cur_xpath[i];
            tie_paths[num_ties].y[i] := cur_ypath[i];
            end;
    end;
```

```
(* score_plus adds a threat value to the score of the current   *)
(* block path if it has not already been added from another box *)

function score_plus(score, x, y : integer) : integer;

    var
            i, k : integer;
    begin
    if threat_decon[x, y].num <> 0 then
            (* there are threat values in this box *)
            begin
            for i := 1 to threat_decon[x, y].num do
                    (* process each threat value for the box *)
                    begin
                    k := threat_decon[x,-y].kind[i];
                    if threat_added.add[k] = 0 then
                            (* score has not yet been added *)
                            begin
                            score := score + threat_decon[x, y].value[i];
                            threat_added.add[k] := 1;
                            threat_added.x[k] := x;
                            threat_added.y[k] := y;
                            end;
                    end;
            end;
    score_plus := score;
    end;

(* score_minus subtracts a threat value from the score of a block *)
(* path before the block path continues in another direction      *)

function score_minus(score, x, y : integer) : integer;

    var
            i, k : integer;
    begin
```

```
(* with a better block path                                    *)

procedure new_best(rank : integer);

    var
            i : integer;
    begin
    if rank = 1 then
            (* new best block path found *)
            begin
            num_ties := 0;
            next_best_score := best_score;
            best_score := cur_score;
            for i := 1 to x_max do
                    (* replace best path with current path and *)
                    (* replace next best with old best path    *)
                    begin
                    next_best_xpath[i] := best_xpath[i];
                    next_best_ypath[i] := best_ypath[i];
                    best_xpath[i] := cur_xpath[i];
                    best_ypath[i] := cur_ypath[i];
                    end;
            end
    else
            (* new next best path found *)
            begin
            next_best_score := cur_score;
            for i := 1 to x_max do
                    (* replace next best path with current path *)
                    begin
                    next_best_xpath[i] := cur_xpath[i];
                    next_best_ypath[i] := cur_ypath[i];
                    end;
            end;
    end;
```

```
                    threat[19, 7] := -1;
                    threat[19, 8] := -1;
                    threat[19, 9] := -1;
                    threat[19, 10] := -1;
                    threat[20, 1] := -1;
                    threat[20, 2] := -1;
                    threat[20, 3] := -1;
                    threat[20, 4] := -1;
                    threat[20, 6] := -1;
                    threat[20, 7] := -1;
                    threat[20, 8] := -1;
                    threat[20, 9] := -1;
                    threat[20, 10] := -1;
                    end;
            end;
    end;

(* save_best_block_paths stores the two best block paths and scores *)

procedure save_best_block_paths;

    var
            . : integer;
    begin
    for i := 1 to x_max do
            begin
            block[1].x[i] := best_xpath[i];
            block[1].y[i] := best_ypath[i];
            block[2].x[i] := next_best_xpath[i];
            block[2].y[i] := next_best_ypath[i];
            end;
    block[1].score := best_score;
    block[2].score := next_best_score;
    end;

(* new_best replaces the current best/next best block path *)
```

```
                threat_decon[20, 1].value[1] := -1;
                threat_decon[20, 2].value[1] := -1;
                threat_decon[20, 3].value[1] := -1;
                threat_decon[20, 4].value[1] := -1;
                threat_decon[20, 6].value[1] := -1;
                threat_decon[20, 7].value[1] := -1;
                threat_decon[20, 8].value[1] := -1;
                threat_decon[20, 9].value[1] := -1;
                threat_decon[20, 10].value[1] := -1;
                end;
        end
else
        begin
        if (x_max = 10) then
                begin
                threat[9, 1] := -1;
                threat[9, 5] := -1;
                threat[10, 1] := -1;
                threat[10, 2] := -1;
                threat[10, 4] := -1;
                threat[10, 5] := -1;
                end
        else
                begin
                threat[16, 10] := -1;
                threat[17, 1] := -1;
                threat[17, 9] := -1;
                threat[17, 10] := -1;
                threat[18, 1] := -1;
                threat[18, 2] := -1;
                threat[18, 8] := -1;
                threat[18, 9] := -1;
                threat[18, 10] := -1;
                threat[19, 1] := -1;
                threat[19, 2] := -1;
                threat[19, 3] := -1;
```

```
for i := 1 to x_max do
        begin
        best_xpath[i] := 0;
        best_ypath[i] := 0;
        end;
(* mark all boxes in grid that cannot possibly *)
(* lead to the goal as not used                *)
if option = DECONFLICT then
        begin
        if (x_max = 10) then
                begin
                threat_decon[9, 1].value[1] := -1;
                threat_decon[9, 5].value[1] := -1;
                threat_decon[10, 1].value[1] := -1;
                threat_decon[10, 2].value[1] := -1;
                threat_decon[10, 4].value[1] := -1;
                threat_decon[10, 5].value[1] := -1;
                end
        else
                begin
                threat_decon[16, 10].value[1] := -1;
                threat_decon[17, 1].value[1] := -1;
                threat_decon[17, 9].value[1] := -1;
                threat_decon[17, 10].value[1] := -1;
                threat_decon[18, 1].value[1] := -1;
                threat_decon[18, 2].value[1] := -1;
                threat_decon[18, 8].value[1] := -1;
                threat_decon[18, 9].value[1] := -1;
                threat_decon[18, 10].value[1] := -1;
                threat_decon[19, 1].value[1] := -1;
                threat_decon[19, 2].value[1] := -1;
                threat_decon[19, 3].value[1] := -1;
                threat_decon[19, 7].value[1] := -1;
                threat_decon[19, 8].value[1] := -1;
                threat_decon[19, 9].value[1] := -1;
                threat_decon[19, 10].value[1] := -1;
```

```
                end;
            end;
    end;

(* conflict_val computes the threat cost between two points *)
(*                                                          *)
(* NOTE  except for a few differences, this routine is from *)
(*       program PATH_FINDER written by Carl Lizza and is   *)
(*       used here with his permission                      *)

function conflict_val(x1, y1, x2, y2 : real) : real;

    var
        i : integer;
        cost, p1_to_p2, x1_to_x2, y1_to_y2, z, center_to_line,
          p1_to_center, p2_to_center : real;
    begin
    cost := 0.0;
    p1_to_p2 := distance(x1, y1, x2, y2);
    x1_to_x2 := x1 - x2;
    y1_to_y2 := y1 - y2;
    z := x1_to_x2 * y1 - y1_to_y2 * x1;
    for i := 1 to threat_num do
        (* check points against each threat *)
        begin
        center_to_line := abs(y1_to_y2 * threat_in.cx[i] -
                              x1_to_x2 * threat_in.cy[i] + z) / p1_to_p2;
        p1_to_center := distance(x1, y1, threat_in.cx[i], threat_in.cy[i]);
        p2_to_center := distance(x2, y2, threat_in.cx[i], threat_in.cy[i]);
        if center_to_line < threat_in.radius[i] then
            (* does threat cover an endpoint *)
            begin
            if ((p2_to_center < threat_in.radius[i]) and
                (p1_to_center >= threat_in.radius[i])) or
               ((x1 = 0.0) and ( p1_to_center <= threat_in.radius[i])) then
                    cost := cost + threat_in.value[i]
```

```
                else if (p2_to_center <= sqrt(p1_to_center ** 2 + p1_to_p2 ** 2)) and
                        (p1_to_center <= sqrt(p2_to_center ** 2 + p1_to_p2 ** 2)) and
                        (p1_to_center >= threat_in.radius[i]) then
                        cost := cost + threat_in.value[i];
                end;
        end;
    conflict_val := cost;
    end;

(* save_path stores the current path so far *)
(* into a temporary array                  *)

procedure save_path(path_len, path_to, path_from : integer);

    var
        i : integer;
    begin
    for i := 1 to path_len do
        begin
        temp_path.xpt[path_to, i] := path.xpt[path_from, i];
        temp_path.ypt[path_to, i] := path.ypt[path_from, i];
        temp_path.conflict[path_to, i] := path.conflict[path_from, i];
        temp_path.len[path_to, i] := path.len[path_from, i];
        end;
    end;

(* restore_path places a stored path back into the current path array *)

procedure restore_path(path_len : integer);

    var
        i, j : integer;
    begin
    for i := 1 to pts_on_line do
        begin
        for j := 1 to path_len do
```

```
                    begin
                    path.xpt[i, j] := temp_path.xpt[i, j];
                    path.ypt[i, j] := temp_path.ypt[i, j];
                    path.conflict[i, j] := temp_path.conflict[i, j];
                    path.len[i, j] := temp_path.len[i, j];
                    end;
            end;
    end;

(* compute_flight_paths stores all flight paths within a *)
(* block path including their costs and lengths          *)

procedure compute_flight_paths;

    var
            i, j, k : integer;
            dist, score, temp_dist, temp_score : real;
    begin
    for i := 1 to pts_on_line do (* first leg *)
            begin
            dist := distance(first_xpt, first_ypt, next_pt.xcoor[1, i],
                             next_pt.ycoor[1, i]);
            score := conflict_val(first_xpt, first_ypt, next_pt.xcoor[1, i],
                                  next_pt.ycoor[1, i]);
            path.xpt[i, 1] := first_xpt;
            path.ypt[i, 1] := first_ypt;
            path.conflict[i, 1] := score;
            path.len[i, 1] := dist;
            end;
    for i := 1 to num_div - 2 do (* middle legs *)
            begin
            for j := 1 to pts_on_line do (* for each of the points on a division *)
                    begin
                    temp_score := 10000.0;
                    temp_dist := 10000.0;
                    for k := 1 to pts_on_line do (* for each of the points on the next division *)
```

```
begin
dist := distance(next_pt.xcoor[i, k],
                 next_pt.ycoor[i, k],
                 next_pt.xcoor[i + 1, j],
                 next_pt.ycoor[i + 1, j]) +
        path.len[k, i];
score := conflict_val(next_pt.xcoor[i, k],
                      next_pt.ycoor[i, k],
                      next_pt.xcoor[i + 1, j],
                      next_pt.ycoor[i + 1, j])
                      + path.conflict[k, i];
if score <= temp_score then
        begin
        if score = temp_score then
                (* tie score - check distance *)
                begin
                if dist < temp_dist then
                        begin
                        path.xpt[j, i + 1] := next_pt.xcoor[i, k];
                        path.ypt[j, i + 1] := next_pt.ycoor[i, k];
                        path.conflict[j, i + 1] := score;
                        path.len[j, i + 1] := dist;
                        temp_score := score;
                        temp_dist := dist;
                        save_path(i, j, k);
                        end;
                end
        else
                (* better score - save path *)
                begin
                path.xpt[j, i + 1] := next_pt.xcoor[i, k];
                path.ypt[j, i + 1] := next_pt.ycoor[i, k];
                path.conflict[j, i + 1] := score;
                path.len[j, i + 1] := dist;
                temp_score := score;
                temp_dist := dist;
```

```
                                        save_path(i, j, k);
                                        end;
                                    end;
                                end; (* k loop *)
                            end; (* j loop *)
                    restore_path(i);
                    end; (* i loop *)
        for i := 1 to pts_on_line do (* last leg *)
                    begin
                    dist := distance(next_pt.xcoor[num_div - 1, i], next_pt.ycoor[num_div - 1, i],
                                last_xpt, last_ypt) + path.len[i, num_div - 1];
                    score := conflict_val(next_pt.xcoor[num_div - 1, i],
                                next_pt.ycoor[num_div - 1, i], last_xpt,
                                last_ypt) + path.conflict[i, num_div - 1];
                    path.xpt[i, num_div] := next_pt.xcoor[num_div - 1, i];
                    path.ypt[i, num_div] := next_pt.ycoor[num_div - 1, i];
                    path.conflict[i, num_div] := score;
                    path.len[i, num_div] := dist;
                    end;
        end;

(* figure_pts computes all the points on each division *)
(* line for each box in the block path                *)

procedure figure_pts;

        var
                i, j, box_size : integer;
                min_pt, max_pt, segment : real;
        begin
        box_size := x_limit div x_max;
        for i := 1 to x_max - 1 do
                (* find the min and max points for each box division *)
                begin
                if best_ypath[i] > best_ypath[i + 1] then
                        begin
```

```
                max_pt := best_ypath[i] * box_size;
                min_pt := max_pt - (2 * box_size);
                end;
        if best_ypath[i] < best_ypath[i + 1] then
                begin
                max_pt := best_ypath[i + 1] * box_size;
                min_pt := max_pt - (2 * box_size);
                end;
        if best_ypath[i] = best_ypath[i + 1] then
                begin
                max_pt := best_ypath[i] * box_size;
                min_pt := max_pt - box_size;
                end;
        segment := (max_pt - min_pt) / (pts_on_line - 1);
        for j := 1 to pts_on_line do
                (* compute each point for a box division *)
                begin
                if ((num_div = 10) or (x_max = 20)) then
                        begin
                        next_pt.xcoor[i, j] := i * box_size;
                        next_pt.ycoor[i, j] := min_pt + ((j - 1) * segment);
                        end
                else
                        begin
                        next_pt.xcoor[i*2-1, j] := (i * box_size) - (box_size div 2) ;
                        next_pt.ycoor[i*2-1, j] := ((best_ypath[i]*box_size)-box_size)
                                +((j-1)*(x_max/(pts_on_line-1)));
                        next_pt.xcoor[i*2, j] := i * box_size;
                        next_pt.ycoor[i*2, j] := min_pt+((j-1)*segment);
                        end;
                end;
        end;
if ((num_div = 20) and (x_max = 10)) then
        begin
        for j := 1 to pts_on_line do
                begin
```

```
                    next_pt.xcoor[num_div-1, j] := (x_max*box_size)-(box_size div 2);
                    next_pt.ycoor[num_div-1, j] := ((best_ypath[x_max]*box_size)-box_size)
                            + ((j-1)*(x_max/(pts_on_line-1)));
                    end;
            end;
    end;


    begin  (* main section of exhaust *)
    path_num := path_num + 1;
    figure_pts;
    compute_flight_paths;
    compute_best_flight_path(inx);
    save_flight(path_num, inx);
    end;

(* next_flight_path transfers the next path into best_path *)
(* for processing by exhaust                               *)

procedure next_flight_path;

    var
            i : integer;
    begin
    for i := 1 to x_max do
            begin
            best_xpath[i] := next_best_xpath[i];
            best_ypath[i] := next_best_ypath[i];
            end;
    end;

(* output_paths chooses the best flight path and its *)
(* block path and prints the final results           *)

procedure output_paths;
```

```
var
        i, inx : integer;
begin
inx := 1;
if (flight[inx + 1].cost < flight[inx].cost) or
      ((flight[inx + 1].cost = flight[inx].cost) and
   (flight[inx + 1].len < flight[inx].len)) then
        inx := 2;
tot_cpu_time := clock - tot_cpu_time;  (* system clock *)
(* print block path and score *)
writeln(pathout);
writeln(pathout, 'BLOCK PATH');
writeln(pathout);
writeln(pathout, '         X         Y');
writeln(pathout);
for i := 1 to x_max do
        writeln(pathout, block[inx].x[i], block[inx].y[i]);
writeln(pathout);
writeln(pathout, 'SCORE = ', block[inx].score : 4);
writeln(pathout);
(* print flight path, cost, and length *)
writeln(pathout);
writeln(pathout, 'FLIGHT PATH');
writeln(pathout);
writeln(pathout, '         X         Y');
writeln(pathout);
for i := 1 to num_div do
        writeln(pathout, flight[inx].x[i] :10:2, flight[inx].y[i] :10:2);
writeln(pathout, last_xpt :10:2, last_ypt :10:2);
writeln(pathout);
writeln(pathout, 'COST = ', flight[inx].cost : 6:2);
writeln(pathout);
writeln(pathout, 'LENGTH = ', flight[inx].len : 6:2);
writeln(pathout);
writeln(pathout, 'CPU TIME(msec) = ', tot_cpu_time);
end;
```

```
begin  (* main section of hplan *)
initialize;
read_data;
process_threats;
block_paths;
exhaust;
next_flight_path;
exhaust;
output_paths;
end.
```

APPENDIX B

```
(* program threat_bldr builds output file datain for use by program *)
(* hplan and contains run options and a random threat environment   *)
(*                                                                  *)
(*    NOTE  except for a few differences, this program was written  *)
(*          by Carl Lizza and is used here with his permission      *)

program threat_bldr(input, output, datain);

    const
            pi = 3.1415927;
            x_limit = 100;
            y_limit = 50;

    type
            block_size = (BIG, SMALL);
            cost_option = (ALL, DECONFLICT);
            coverage = (CENTER, TOTAL);

    var
            i, j, nr_threats, count, x_max, y_max,
              seed, num_div, pts_on_line, cost : integer;
            grid_area, total_density, radius,
              threat_density, x_coord, y_coord : real;
            type_path : block_size;
            option : cost_option;
            box_coverage : coverage;
            datain : text;

(* rand computes a random number *)

function rand(var seed : integer) : real;
```

```
begin
seed := ((25173 * seed) + 13849) mod 65536;
rand := seed / 65536;
end;


begin  (* main section of threat_bldr *)
rewrite(datain);
writeln('ENTER TYPE OF BLOCK PATH -- BIG or SMALL');
readln(type_path);
writeln(datain, type_path);
writeln('ENTER COST OPTION -- ALL or DECONFLICT');
readln(option);
writeln(datain, option);
writeln('ENTER BOX THREAT COVERAGE -- CENTER or TOTAL');
readln(box_coverage);
writeln(datain, box_coverage);
writeln('ENTER MAXIMUM XY BLOCK COORDINATES -- XMAX YMAX');
readln(x_max, y_max);
writeln(datain, x_max, y_max);
writeln('ENTER NUMBER OF DIVISIONS AND THE NUMBER OF');
writeln('POINTS FOR EACH DIVISION -- NUMDIV PTS');
readln(num_div, pts_on_line);
writeln(datain, num_div, pts_on_line);
writeln('ENTER SEED');
readln(seed);
grid_area := x_limit * y_limit;
writeln('ENTER NUMBER OF THREAT CATEGORIES');
readln(nr_threats);
writeln('ENTER TOTAL THREAT DENSITY');
readln(total_density);
for i := 1 to nr_threats do
        begin
        writeln('FOR THREAT CATEGORY', i : 3, ' ENTER RADIUS, COST, DENSITY');
        readln(radius, cost, threat_density);
        count := trunc(total_density * threat_density * grid_area /
```

```
                        (pi * radius ** 2));
        for j := 1 to count do
                begin
                x_coord := x_limit * rand(seed);
                y_coord := y_limit * rand(seed);
                writeln(datain, x_coord, y_coord, cost, radius);
                end;
        end;
writeln(datain, -1, -1, -1, -1);
end.
```

BIBLIOGRAPHY

1. Fikes, R.E. & N.J. Nilsson, "STRIPS: A New Approach to the Application of Theorem Proving to Problem Solving," Artificial Intelligence, Vol. 2, pp. 189-208, 1971.

2. Fikes, R.E., P.E. Hart, & N.J. Nilsson, "Learning and Executing Generalized Robot Plans," Artificial Intelligence, Vol. 3, No. 4, pp. 251-288, 1972.

3. Georgeff, M.P., "Strategies in Heuristic Search," Artificial Intelligence, Vol. 20, pp. 393-425, 1983.

4. Grove, D., "Documentation for the FPG Model," University of Dayton, 1980.

5. Lizza, C.S., "Generation of Flight Paths Using Heuristic Search," Wright State University, 1984.

6. McLaughlin, R.G., "Description and Use of SNOOPER III, A Model for Determining the Strategy Needed Over Optimum PEnetration Routes," Cornell Aeronautical Laboratory, 1971.

7. Nilsson, N.J., Principles of Artificial Intelligence, Tioga, Palo Alto, Calif., 1980.

8. Ralston, A., Encyclopedia of Computer Science, Van Nostrand Reinhold, New York, 1976.

BIBLIOGRAPHY (CONTINUED)

9. Rich, E., Artificial Intelligence, McGraw-Hill, New York, 1983.

10. Sacerdoti, E.D., "Planning in a Hierarchy of Abstraction Spaces," Artificial Intelligence, Vol. 5, pp. 115-135, 1974.

11. Sacerdoti, E.D., A Structure for Plans and Behavior, Elsevier, New York, 1977.

12. Stefik, M., "Planning with Constraints (MOLGEN: Part 1)," Artificial Intelligence, Vol. 16, No. 2, pp. 111-139, 1981.

13. Stefik, M., "Planning and Meta-Planning (MOLGEN: Part 2)," Artificial Intelligence, Vol. 16, No. 2, pp. 141-169, 1981.

14. Wilkins, D., "Using Patterns and Plans in Chess," Artificial Intelligence, Vol. 14, pp. 165-203, 1980.

END

FILMED

8-85

DTIC